**MASTER EN SCIENCES INFORMATIQUES**

# Conception d'un Interpréteur Prolog sur base de la Machine Abstraite de Warren

CAMMARATA, Frédéric; WAERENBURGH, Michaël

*Award date:*
1994

*Awarding institution:*
Universite de Namur

**Facultés Universitaires Notre Dame de La Paix**
**Namur**
**Institut d'Informatique**

# Conception d'un Interpréteur Prolog sur base de la Machine Abstraite de Warren

**Promoteur : B. Le Charlier**

Mémoire présenté pour l'obtention du grade
de Licencié et Maître en
informatique
par :

**Cammarata Frédéric**
**Waerenburgh Michaël**

**Année académique 1993-1994**

Nous remercions Monsieur Le Charlier de ses nombreux conseils ainsi que de nous avoir permis de réaliser ce mémoire ensemble.

Merci également à nos familles et à nos proches pour leur soutien.

## Résumé

Une approche possible dans la conception d'un compilateur Prolog réside dans la notion de machine virtuelle. Depuis sa définition par D.H.Warren, la machine virtuelle connue sous le nom de Machine Abstraite de Warren a été abondamment reprise dans d'autres travaux et constitue une solide référence en matière de compilation. Sur base de l'ouvrage de Aït-Kaci, "W.A.M., a tutorial reconstruction", nous réalisons l'implémentation de cette machine. Après un bref récapitulatif des notions de Prolog pur, nous exposons les représentations introduites pour la machine. Ensuite, nous détaillons les diverses étapes de construction. Une spécification formelle des instructions est suggérée et complétée par la correction de certaines d'entre elles. Nous clôturons par un aperçu des optimisations envisageables sur cette machine suivi d'une description de l'important travail d'analyse nécessaire à la génération des instructions virtuelles.



## Abstract

A possible approach in the conception of a Prolog compilator consists of the notion of the virtual machine. Since its definition by D.H. Warren, the virtual machine known as " Warren's Abstract Machine" has been widely used in other works and constitutes a solid reference in the field of compilation. On the basis of Aït-Kaci's document, "W.A.M., a tutorial reconstruction", our main interest is the implementation of this machine. After a brief summary of Pure Prolog notions, we will expose the representations introduced for the machine. Afterwards we will detail the different steps of the construction. A formal specification of the instructions is suggerated and achieved by the correctness of some of them. We will finish with a survey of the possible optimisations concerning this machine followed by a description of the important work of the analysis which is essential for the code generation.

# Sommaire

# Introduction

Depuis plusieurs années, l'essor des systèmes Prolog est tel que de plus en plus de domaines d'applications s'ouvrent à eux. Ces systèmes s'inscrivent aussi bien dans le cadre de la programmation logique, que dans ceux des systèmes experts, des langages parallèles et de la programmation sous contraintes.

D'autre part, l'intérêt de disposer d'un compilateur pour un tel langage réside avant tout dans la possibilité d'augmenter la vitesse d'exécution d'un ou d'une partie de programme que l'on sait être quasi-définitive. A l'inverse, l'interprétation, plus lente, favorise par contre la construction et la mise au point interactive de programmes.

La plupart des solutions proposées pour ces sytèmes Prolog travaillent à partir de la notion de machine virtuelle dans un souci évident de portabilité et de rentabilité. D.H. Warren signale cependant que le code généré peut être traité sous différentes formes : par un émulateur, par génération de l'assembleur ou directement par un processeur dédié.
Nous retiendrons cette approche de conception d'un compilateur à partir de la notion de machine virtuelle et en particulier celle de D.H. Warren connue sous le nom de Machine Abstraite de Warren ( W.A.M. ). En effet, depuis sa définition, cette machine virtuelle a été abondamment reprise dans d'autres travaux et constitue donc une solide référence en matière de compilation.

Nous avons souhaité implémenter cette machine virtuelle en langage procédural. Sur base de l'ouvrage de Aït-Kaci [AK91], nous détaillons pas à pas les diverses étapes de construction d'une telle machine. A chaque étape, nous apportons une spécification formelle des instructions proposées ainsi qu'une démonstration de la correction de celles qui nous semblent les plus fondamentales. De plus, nous essayons de justifier les différents choix jalonnant chaque étape. Nous clôturons chaque étape en l'illustrant de divers exemples ainsi que du code source des instructions WAM envisagées. Le lecteur peut cependant omettre de lire la partie du code source sans que cela ne porte préjudice à la compréhension

de ce travail.

Celui-ci s'articulera de la façon suivante :

**Chapitre 1 : Notions de Prolog pur**
Nous rappellons tout d'abord les principes de Prolog classique tels que la substitution et l'unification. Puis, nous définissons succintement la sémantique opérationnelle et nous décrivons la SLD-résolution, sémantique utilisée, suivie d'un exemple.

**Chapitre 2 : Représentations dans la Machine Abstraite de Warren**
Suivant une classification des systèmes Prolog classiques, nous envisageons les choix de la Machine Abstraite ainsi que leurs justifications. Ensuite, nous suggèrons une formalisation de ces représentations. Celle-ci s'avérera utile lors de la spécification et de la correction des instructions. Nous terminons en décrivant l'architecture de la mémoire.

**Chapitre 3 : Construction de la Machine Abstraite de Warren**
Cette construction s'effectuera en quatre étapes. La première introduit l'unification entre deux termes, la seconde entre une question à un but et un ensemble de faits, la suivante entre une question à plusieurs buts et un ensemble de clauses sans backtracking, et la dernière complète la machine en lui apportant le backtracking.
Tout au long de cette construction, nous tâcherons de spécifier formellement les instructions grâce au formalisme introduit dans le chapitre précédent. Nous y démontrerons également la correction de l'algorithme d'unification et de certaines autres instructions. Nous détaillerons explicitement la production de séquences d'instructions en y dévoilant les structures récursives sous-jacentes existantes.

**Chapitre 4 : Prolongements**
Ce travail tout en étant complet offre encore de nombreuses directions de recherche et d'amélioration. Dans un premier temps, nous passons en revue les optimisations envisageables sur la Machine Abstraite de Warren. Dans un second temps, nous aborderons les avantages que l'interprétation abstraite pourrait y apporter.

**Chapitre 5 : L'analyseur syntaxique**
Dans la conception d'un interpréteur, une importante partie du travail est l'analyse du programme et des données avant d'en générer le code d'instructions virtuelles WAM. D'abord, les structures de données utilisées dans cette optique seront explicitées. Puis, les analyses seront expliquées à l'aide d'exemples. Finalement, nous expliquons brièvement l'utilisation des divers interpréteurs.

# Chapitre 1

# Notions de Prolog Pur

Nous allons rappeler quelques définitions qui nous semblent importantes pour la compréhension de notre travail. Ce chapitre ne se veut pas du tout exhaustif. Nous préférons renvoyer le lecteur qui souhaite de plus amples informations à l'ouvrage de Lloyd [Llo87].

## 1.1 Syntaxe de Prolog

Commençons par aborder les divers types de données en Prolog. Un terme y est défini comme étant :

- soit une constante i.e. un identifiant commençant par une minuscule ( ex. : a, toto, b212, ... )
- soit une variable i.e. un identifiant commençant par une majuscule ( ex. : W, X, Y, ... )
- soit une construction de la forme $f(t_1, ..., t_n)$ dont l'identifiant commence par une minuscule, $f$ est le foncteur d'arité $n$ et les $t_1, ..., t_n$ sont des termes déjà construits.

Nous pouvons ainsi écrire de manière syntaxique :

⟨ terme ⟩ ::= ⟨ constante ⟩ | ⟨ variable ⟩ | ⟨ construction ⟩
⟨ constante ⟩ ::= ⟨ minuscule ⟩ | ⟨ minuscule ⟩ { ⟨ lettre ⟩ }
⟨ variable ⟩ ::= ⟨ majuscule ⟩ | ⟨ majuscule ⟩ { ⟨ lettre ⟩ }
⟨ construction ⟩ ::= ⟨ foncteur ⟩ ( ⟨ terme ⟩ { , ⟨ terme ⟩ } )
⟨ foncteur ⟩ ::= ⟨ constante ⟩
⟨ minuscule ⟩ ::= ⟨ a ⟩ | ... | ⟨ z ⟩
⟨ majuscule ⟩ ::= ⟨ A ⟩ | ... | ⟨ Z ⟩
⟨ lettre ⟩ ::= ⟨ minuscule ⟩ | ⟨ majuscule ⟩

Décrivons maintenant ce que nous entendons par un programme Prolog. Un tel programme "P" est constitué de procédures "pr". Celles-ci sont formées d'une ou plusieurs clauses "c" qui ont le même nom. Les clauses sont constituées d'une tête et d'un corps. Ce corps sera soit vide soit constitué d'une suite d'atomes. Nous pouvons donc écrire de manière syntaxique que :

⟨ P ⟩ ::= {⟨pr⟩}
⟨pr⟩ ::= ⟨c⟩ | ⟨c⟩ ⟨pr⟩
⟨c⟩ ::= ⟨tete⟩: −⟨corps⟩.
⟨tete⟩ ::= ⟨atome⟩
⟨corps⟩ ::= ⟨⟩ | ⟨atome⟩{,⟨atome⟩}
⟨atome⟩ ::= ⟨construction⟩

## 1.2 L'unification

En programmation logique, le but principal d'un système est de générer des liaisons. Celles-ci sont générées par l'unification en Prolog.
Détaillons cette notion d'unification.

**Définition :**

Une **substitution** $\sigma$ est un ensemble fini de couples de la forme $\{x_1/t_1, ..., x_n/t_n\}$ où les $x_1, ..., x_n$ sont des variables distinctes et les $t_1, ..., t_n$ sont des termes, de plus $\forall i\ x_i \neq t_i$. Si on applique la substitution $\sigma = \{x_1/t_1, ..., x_n/t_n\}$ au terme $t$ , le terme $t\sigma$ est obtenu en remplaçant simultanément toutes les occurrences de $x_i$ par $t_i$ pour chaque i.

**Définition :**

$t'$ est appelé **instance** de $t$ si et seulement s'il existe une substitution $\sigma$ telle que $t' = t\sigma$ .

**Définition :**

Soient $\theta = \{y_1/u_1, ..., y_m/u_m\}$ et $\sigma = \{x_1/t_1, ..., x_n/t_n\}$ deux subsitutions.

La **composition** $\theta\sigma$ de $\theta$ et de $\sigma$ est la substitution obtenue à partir de l'union des ensembles $\theta$ et $\sigma$ , $\{x_1/t_1, ..., y_m/u_m\}$ en éliminant tous les couples $y_i/u_i$ pour lesquels $y_i = u_i\sigma$ et tous les couples $x_j/t_j$ pour lesquels $x_j \in \{y_1, ..., y_m\}$.

**Définition :**

La substitution produite à partir de l'ensemble vide est appelée substitution **identité** $\varepsilon$.

**Propriétés :**

Soient $\theta, \sigma, \gamma$ des substitutions.

| | | |
|---|---|---|
| neutre | $\theta\varepsilon = \theta = \varepsilon\theta$ | |
| composition | $(t\theta)\sigma = t(\sigma\theta)$ | $\forall t$ terme |
| associativité | $\theta(\sigma\gamma) = (\theta\sigma)\gamma$ | |

La preuve de ces propriétés se trouve dans [Llo87].

**Définitions :**

Une substitution $\sigma$ est appelée **unificateur** ( elle réalise donc l'unification ) de deux termes $t_1$ et $t_2$ s'il existe $t$ tq $t_1\sigma = t = t_2\sigma$ .

Un unificateur $\sigma$ de $t_1$ et $t_2$ est appelé " **plus général** " ( et noté " mgu " ) si pour tout unificateur $\theta$ de $t_1$ et $t_2$ , il existe une subsitution $\tau$ telle que $\theta = \sigma\tau$.

Un algorithme, appelé **algorithme d'unification**, réalise l'unification de deux termes $t_1$ et $t_2$ et génère le mgu correspondant. L'unification réalisée nous fournit l'instanciation des deux termes.

Spécifions l'algorithme de base.

- Pré : $t_1$ et $t_2$ sont les termes à unifier.
- Post :

si $t_1$ et $t_2$ ne sont pas unifiables
alors fail
sinon $\sigma$ = mgu ( $t_1$ , $t_2$ )

Grâce à la méthode du programme auxiliaire, nous pouvons généraliser cet algorithme en considérant l'ensemble des termes qu'il reste à unifier et la substitution intermédiaire déjà réalisée.
En initialisant la substitution intermédiaire à l'identité et l'ensemble des termes aux deux termes $t_1$ et $t_2$ , il est aisé de vérifier que nous retrouvons bien l'algorithme de base.

Spécifions l'algorithme général .

- Pré : $\sigma$ , la substitution intermédiaire et E, l'ensemble des couples de termes à unifier.
- Post :

  si E n'est pas unifiable
  alors fail
  sinon $\sigma\sigma'$ est la nouvelle substitution où $\sigma'$ = mgu ( E ).

Nous pouvons détailler cet algorithme sous forme d'un ensemble de règles de transitions.

1. $\langle \sigma, \{\} \rangle \longmapsto \sigma$

2. $X \neq t \wedge X \notin Var(t)$
   $\langle \sigma, X = t :: E \rangle \longmapsto \langle \sigma\{X/t\}, E\{X/t\} \rangle$

3. $X \neq t \wedge X \in Var(t)$
   $\langle \sigma, X = t :: E \rangle \longmapsto$ fail

4. $\langle \sigma, X = X :: E \rangle \longmapsto \langle \sigma, E \rangle$

5. $\langle \sigma, t = X :: E \rangle \longmapsto \langle \sigma, X = t :: E \rangle$

6. $(f \neq g) \vee (n \neq m)$
   $\langle \sigma, f(t_1, ..., t_n) = g(u_1, ..., u_m) :: E \rangle \longmapsto$ fail

7. $\langle \sigma, f(t_1, ..., t_n) = f(u_1, ..., u_n) :: E \rangle \longmapsto \langle \sigma, t_1 = u_1 :: ... :: t_n = u_n :: E \rangle$

Remarquons que les prémisses des règles 2. et 3. recouvrent le principe de l'**Occur-Check**. Si la prémisse de la règle 3. est vérifiée, respectivement la prémisse de la règle 2. mise en défaut, on dit qu'il y a occur-check. En d'autres mots, ce test d'occurrence vérifie qu'une variable ne soit pas unifiée à un terme la contenant.

Notons aussi que cet algorithme n'est pas déterministe. En effet, nous pouvons choisir n'importe quelle équation de l'ensemble $E$ et lui appliquer l'une des sept règles (n'importe laquelle pour autant que cette règle s'applique à l'équation). Illustrons ce fait sur un exemple.

Soit à unifier les deux termes suivants : f(a,g(X)) et f(Y,g(h(Y)). Nous avons donc au départ la situation suivante :

$$\langle \{ \}, \{ \text{f(a, g(X))} = \text{f(Y, g(h(Y)))} \} \rangle$$

règle 7.

$$\langle \{ \}, \{ \text{a} = \text{Y}, \text{g(X)} = \text{g(h(Y))} \} \rangle$$

règle 5. La règle 7 peut aussi s'appliquer

$$\langle \{ \}, \{ \text{Y} = \text{a}, \text{g(X)} = \text{g(h(Y))} \} \rangle$$

règle 2. ( $\text{Y} \neq \text{a}$ et $\text{Y} \notin Var\,(\text{a})$ )

$$\langle \{\text{Y} \mid \text{a}\}, \{\text{g(X)} = \text{g(h(a))}\} \rangle$$

règle 7.

$$\langle \{\text{Y} \mid \text{a}\}, \{\text{X} = \text{h(a)}\} \rangle$$

règle 2.

$$\langle \{\text{X} \mid \text{h(a)}, \text{Y} \mid \text{a}\}, \{ \} \rangle$$

règle 1.

$$\{\text{X} \mid \text{h(a)}, \text{Y} \mid \text{a}\}$$

## 1.3 Sémantique

Notre but ici n'est pas de donner l'ensemble des sémantiques liées à Prolog. Il s'agit seulement de mieux définir le cadre dans lequel ce travail s'inscrit. Rappelons qu'une sémantique donne une signification au programme. Dans la littérature, il existe plusieurs sémantiques attachées à Prolog. Parmi les plus connues, citons la sémantique logique, aussi appelée déclarative, et la sémantique opérationnelle. La sémantique logique est basée sur la sémantique du modèle de la logique des prédicats du premier ordre et sur le plus petit modèle d'Herbrand. La sémantique opérationnelle est, quant à elle, un moyen de décrire la signification d'un programme à l'aide de procédures, la signification d'un programme étant l'ensemble des buts clos, instanciés de la question. Celle-ci est résolue par le programme en utilisant un interpéteur abstrait comme celui décrit ci-dessous [Sha 86] :

**Input :** Un programme logique $P$.
Un but $G$.

**Output:** $G\theta$ s'il s'agit d'une instance de $G$ déduit de $P$,
ou *échec* s'il s'est produit une erreur,
où $\theta$ est le mgu de G par rapport à P.

**Algorithme:**

Initialiser la résolvante $G$.
Tant que la résolvante n'est pas vide faire
Choisir un but $A$ de la résolvante et une clause
renommée $A' \leftarrow B_1, B_2, ..., B_n, n \geq 0$, de $P$
tel que $A$ et $A'$ s'unifient avec comme mgu $\theta$
( sortir s'il n'existe pas un tel but ou une telle clause ).
Enlever $A$ de la résolvante et ajouter $B_1, B_2, ...,$ et $B_n$
à la résolvante.
Appliquer $\theta$ à la résolvante et à $G$.
Si la résolvante est vide, sortir $G$ sinon sortir *échec*.

Ce travail s'inscrit dans le cadre d'une telle sémantique.

## 1.4 SLD-résolution

Un système Prolog peut être vu comme un démonstrateur de théorème. Cela consiste à poser des questions dans le cadre d'un univers défini sous une forme clausale[1]. La réponse

[1]En Prolog, on parle de clauses de Horn.

à une question s'effectue par réfutation, c'est-à-dire, qu'en ajoutant la négation de la question au système constitué de clauses, nous pouvons montrer que ce système résultant est non modélisable. Cette réfutation utilise une stratégie particulière de résolution appelée SLD-résolution.[2]

Prolog utilise une telle stratégie. Celle-ci consiste en :

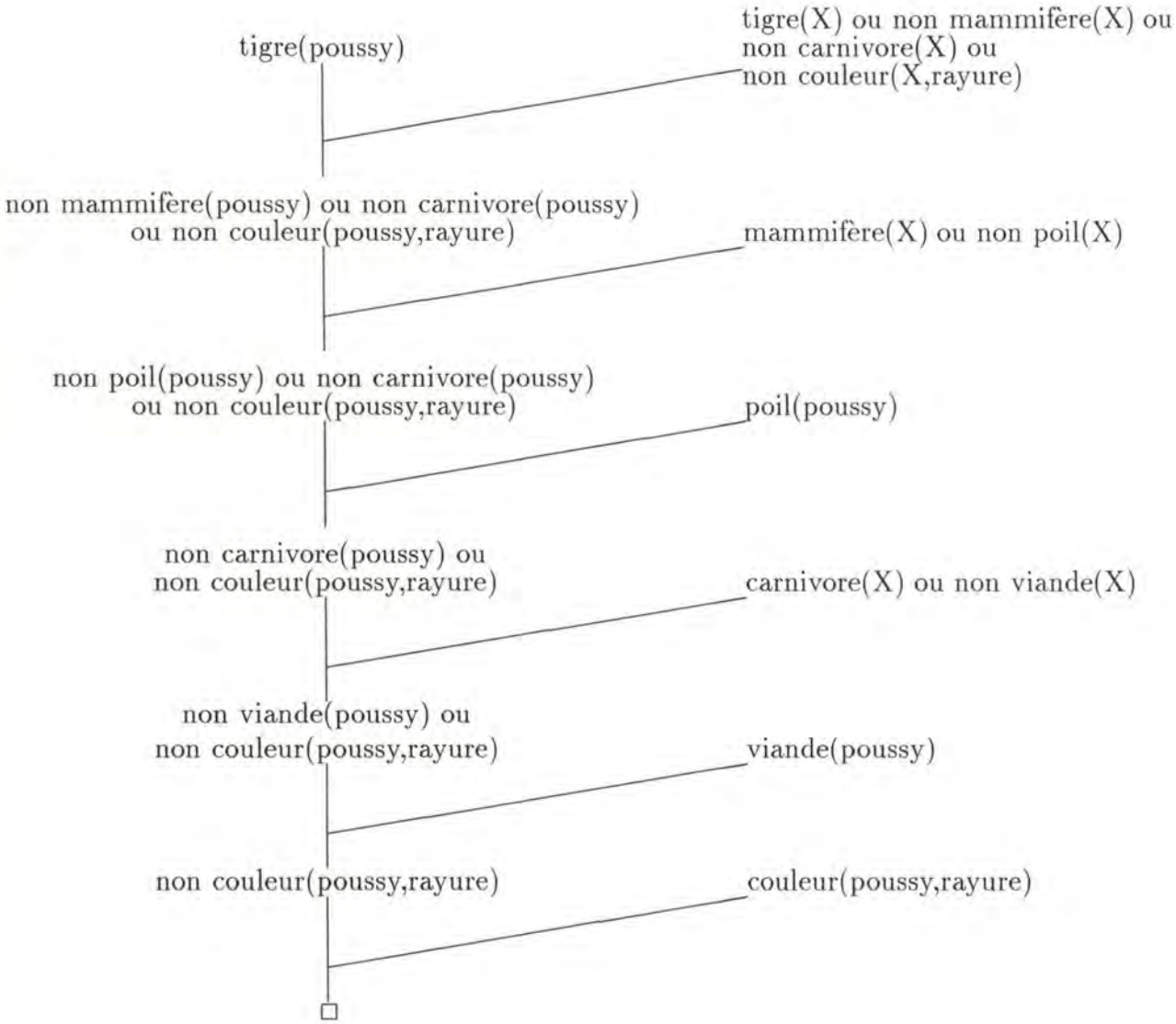
1. dans un premier temps, partir du démenti $C_0$ constitué par la négation de la question;
2. puis, dans un deuxième temps, à construire à chaque étape i ( i > 0 ), une nouvelle résolvante $C_i$ obtenue par la résolution :
   - de la clause $C_{i-1}$ (déduction linéaire)
   - et d'une clause C de la base de connaissance (déduction input)

Illustrons cette stratégie sur un exemple. Considérons la description suivante :

```
félin(poussy).
couleur(poussy,rayure).
viande(poussy).

carnivore(X):-viande(X).
carnivore(X):-yeux_face(X).

mammifère(X):-poil(X).
mammifère(X):-allaite(X).

tigre(X):-mammifère(X),carnivore(X),couleur(X,rayure).
```

Examinons une déduction qui prouve que "poussy" est un tigre. Comme nous pouvons le voir sur la figure suivante, nous partons du démenti de la question et à chaque étape, la résolvante $C_i$ provient de la résolution de la résolvante précédente $C_{i-1}$ avec une clause du programme.

[2]Linear resolution with Selection function for Definite clauses




L'application de cette méthode de résolution nécessite de déterminer :

1. une règle de sélection du littéral;
2. une règle de sélection de clause.

En effet, nous devons choisir un littéral de la résolvante courante auquel va s'appliquer la résolution, ainsi qu'une clause de la base de connaissance.

Nous pouvons dire que quelque soit la règle de sélection du littéral, la complétude de la SLD-résolution est assurée [Llo87]. Le choix de la clause n'interviendra pas non plus si l'on peut garantir que **toutes** les clauses candidates à une unification ont été envisagées, et ceci quelque soit l'ordre.

Comme nous venons de le voir, le choix d'une règle de sélection de clause et d'une règle de sélection de littéral va conditionner pleinement le contrôle des systèmes Prolog.

La plupart de ceux-ci mettent en oeuvre les règles suivantes :

1. Ils choisissent toujours le littéral le plus à gauche dans la résolvante courante.
2. Ils envisagent les clauses dans leur ordre d'apparition dans le programme ( base de connaissance ).

Ainsi, si nous appelons $R_{i-1}$=$B_1, ..., B_n$ la résolvante courante. La règle de sélection du littéral choisira $B_1$ en premier. Si nous supposons que le programme est constitué des clauses : $C_1 = L_1^1, ..., L_{m_1}^1$; ...; $C_n = L_1^n, ..., L_{m_n}^n$, la règle de sélection de la clause choisira $C_1$ pour commencer. Ce sont les règles que nous avons utilisées dans notre exemple. Nous pouvons ajouter que le contrôle de Prolog, défini par le choix de ces règles, peut se représenter par le parcours en profondeur gauche-droite d'un arbre Et/Ou.

# Chapitre 2

# Représentations dans la Machine Abstraite de Warren ( W.A.M. )

Parmi la multiplicité des systèmes Prolog classiques, il est possible d'établir une classification en trois niveaux : l'architecture de la zone de travail, la représentation des termes structurés et les performances de la gestion mémoire. Dans ce chapitre, nous allons principalement nous intéresser à la représentation des termes structurés puis à l'architecture de la zone de travail.

L'unification, opération majeure en Prolog, construit de nouveaux termes par liaison de variables aux termes correspondants. Dans le cas d'un terme simple, on associe directement la variable et le terme. Dans le cas de termes structurés, la liaison peut s'effectuer en **construction**, par création d'une nouvelle instance de terme à partir d'un terme modèle, ou en **décomposition**, par accès à une instance déjà existante. Une liaison s'effectue en construction si le terme devient pour la première fois accessible via une variable, et en décomposition si ce terme était déjà accessible via une autre variable soit en tant que tel soit comme sous-terme d'une instance déjà existante. Illustrons ces liaisons. L'unification de deux termes p(Z,h(W)) et p(h(f(a)),Z) produit la substitution { Z / h(f(a)), W / f(a) } où la variable Z est liée en construction et la variable W est liée en décomposition. En effet, le terme h(f(a)) devient pour la première fois accessible lors de sa liaison avec la variable Z puis W travaille en décomposition.

Il existe en Prolog, deux modes différents de représentation des termes : le partage et la recopie des structures. Tous deux ont pour but de représenter les nouvelles instances de termes générés par l'unification.
Le **partage des structures**, introduit par Boyer et Moore, représente toute instance de

terme structuré sous la forme d'un couple ( squelette, environnement ) où l'environnement décrit les valeurs des variables qui figurent dans le squelette et où le squelette est le modèle de structure du terme.
La **recopie des structures**, introduite par Bruynooghe, génère toute nouvelle instance de terme structuré par recopie du modèle initial. Pour une liaison en construction, le modèle est recopié puis la variable est liée à cette copie tandis que pour une liaison en décomposition, on associe la variable à l'instance du terme déjà recopiée. Utiliser ce mode de représentation en Prolog nécessite une zone mémoire, la pile de recopie dans l'espace de travail de l'interpréteur.

Comme nous le confirme Boizumault dans [Boi88], la machine abstraite de Warren fait le choix d'une représentation des termes par recopie de structures et sa zone de travail est constituée entre autres de deux piles : la pile de recopie ( Heap ) et la pile de restauration ( Trail ). Cette machine gère donc de manière différente les liaisons en construction et celles en décomposition.

Le mécanisme de liaison doit satisfaire les deux impératifs suivants :

- être efficace aussi bien en construction qu'en décomposition afin de ne pas pénaliser l'unification ;
- respecter au mieux la structure de pile afin de ne pas interdire de futures mises à jour.

Dans la pile de recopie, deux sortes de termes doivent être encodés, les variables et les structures de la forme f( $@_1$, ..., $@_n$ ) où f/n est un foncteur n-aire et les $@_i$ sont les adresses des sous-termes dans la pile de recopie. Pour différencier ces termes, un tag explicite signalera le type de la donnée dans la pile.
Une variable sera représentée par une seule cellule composée de ce tag et d'une référence vers une cellule de la pile. Par convention, une variable non liée se référencera elle-même.
Une structure n-aire sera quant à elle représentée par une suite de n+2 cellules dans la pile. Les n+1 dernières cellules seront toujours contigües afin de permettre un accès rapide aux sous-termes d'une structure.
Le format d'une cellule pour une variable est donc

$$\alpha : \boxed{\text{REF} \mid \beta}$$

Si cette cellule se situe à l'adresse $\alpha$ dans la pile, nous introduisons alors la représentation suivante " $\alpha$ repr v " signifiant que la cellule d'adresse $\alpha$ est une représentation de la variable v. Si cette variable est liée, nous observerons alors $\alpha \neq \beta$.
Nous utiliserons dorénavant les notations tag($\alpha$) pour identifier le type (ici REF) et adr($\alpha$)

pour l'adresse (ici $\beta$).
Pour une structure t = f( $t_1$, ..., $t_n$ ), n+2 cellules seront réservées dans la pile. Leur format peut être décrit comme suit : la première cellule d'adresse $\alpha$ est constituée du couple ⟨ STR, $\beta + 1$ ⟩. La seconde, non nécessairement contigüe à la première, comporte le nom du foncteur et son arité n. Les n cellules restantes, obligatoirement contigües à la seconde, seront les références aux n arguments. Chaque cellule est composée d'un couple ⟨ $tag_i$, $\gamma_i$ ⟩. A nouveau, chaque $tag_i$ prend la valeur REf ou STR, respectivement pour une variable ou une structure. $\gamma_i$ est l'adresse du sous-terme dans la pile. La configuration est donc du type :

$\alpha$ :

| STR | $\beta+1$ |
|---|---|

et

| | | |
|---|---|---|
| $\beta$ : | STR | $\beta+1$ |
| $\beta+1$ : | f | n |
| $\beta+2$ : | $tag_1$ | $\gamma_1$ |
| ... | ... | ... |
| $\beta+1+n$ : | $tag_n$ | $\gamma_n$ |

tel que soit $\alpha = \beta$ , soit $\alpha \neq \beta$. De plus, adr($\alpha$) est toujours l'adresse de la cellule qui contient le nom et l'arité de la structure, cette cellule étant toujours suivie des n arguments de la structure.
De façon similaire, nous conviendrons de représenter formellement par " $\alpha$ repr t " le fait que la cellule d'adresse $\alpha$ est l'adresse du début de la représentation de la structure t. Nous pouvons ainsi préciser concernant les sous-termes que

$$( \text{adr}(\alpha)+i ) \text{ repr } t_i, \forall\ i\ 1 \leq i \leq n.$$

Remarquons que, si la valeur de tag ( adr($\alpha$) + $i$ ) est STR — le i$^e$ sous-terme est de type structure — alors $\forall\ i\ 1 \leq i \leq n$, adr( adr($\alpha$) + $i$ ), partie adresse de la cellule située à l'adresse adr($\alpha$) + $i$, référence la cellule comportant le nom et l'arité de la structure, ce qui est bien conforme à la représentation mise en place et aux conventions de formalisme.

Exemple : Soit le terme p( Z, h( Z, W ), f( W ) ), sa représentation est

| | | | | |
|---|---|---|---|---|
| 0 repr h( Z, W ) | 0 : | STR | 1 | |
| | 1 : | h | 2 | |
| | 2 : | REF | 2 | 2 repr Z |
| | 3 : | REF | 3 | |
| 4 repr f( W ) | 4 : | STR | 5 | |
| | 5 : | f | 1 | |
| | 6 : | REF | 3 | 6 repr W |
| 7 repr p( Z, h( Z, W ), f( W ) ) | 7 : | STR | 8 | |
| | 8 : | p | 3 | |
| | 9 : | REF | 2 | 9 repr Z |
| | 10 : | STR | 1 | 10 repr h( Z, W ) |
| | 11 : | STR | 5 | 11 repr f( W ) |

Grâce à ces formats, les données peuvent être stockées dans la pile de recopie, Heap. Formalisons ce Heap.
Notons l'adresse de la première cellule du Heap, base_du_heap. Cette adresse indiquera donc où commence la pile de recopie dans la zone de travail de l'interpréteur. De plus, rien n'impose que cette pile commence au début de cette zone de travail d'où base_du_heap $\geq 0$.

Désignons par I, l'ensemble des adresses des termes construits dans le Heap i.e.
I = { base_du_heap, ..., $H-1$}\{$adr(\alpha) : tag(\alpha)$ =STR et base_du_heap $\leq \alpha \leq H-1$}
où H est un registre global indiquant la première cellule libre du Heap. Cet ensemble est constitué des adresses de toutes les cellules du Heap sauf celles qui contiennent le nom et l'arité de structures. En effet, dans l'exemple, les adresses 1, 5, et 8 ne représentent selon nos conventions aucun terme. Observons qu'à chaque adresse correspond un terme mais qu'un terme peut être représenté par plusieurs adresses. Ainsi, les adresses 4 et 11 sont associées au sous-terme f( W ).

Si l'on note, par convention, $t_\alpha$ le terme représenté à l'adresse $\alpha$, il est évident que

$$\forall \alpha \in I, \alpha \text{ repr } t_\alpha$$

expression que nous réecrirons dorénavant sous la forme

$$\text{Heap repr } (t_\alpha)_{\alpha \in I}$$

Le Heap, pile de recopie, n'est pas la seule zone de travail qui contiendra la représentation de termes. La pile de restauration Trail, la pile d'environnements Stack, et les registres

X, serviront également à ces fins. Il nous a paru normal de généraliser la représentation précédente à ces zones, d'autant qu'aucune modification n'est à apporter.
Notons, pour le Trail (respectivement pour le Stack), le début de sa zone par base_du_trail (respectivement par base_du_stack) et le registre global indiquant la première cellule libre de la pile par TR (respectivement par B). Nous décrirons plus tard l'utilité de ces différentes zones.
La machine abstraite de Warren utilise également des registres X pour stocker les termes intermédiaires. Nous conviendrons de noter

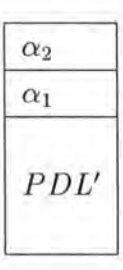
$$X_i \text{ repr } t_{X_i}$$

où $t_{X_i}$ est le terme représenté dans le registre $X_i$, ce qui ne changera rien à nos conventions précédentes.

Avant de passer à l'architecture de la mémoire WAM, détaillons une autre pile d'adresses d'une grande utilité.
Nous avons vu au chapitre précédent qu'unifier deux termes $t_1$ et $t_2$ revenait à résoudre l'équation $t_1 = t_2$. L'algorithme général travaillait sur un ensemble fini d'équations des termes à unifier : $E = \{t_1^1 = t_2^1, ..., t_1^n = t_2^n\}$.
L'algorithme d'unification envisagé dans la machine abstraite de Warren utilise également cette méthode grâce à une pile locale PDL. Nous conviendrons que

PDL repr E

En effet, si E est vide alors PDL l'est également sinon comme E est non vide,
$E = \{ t_1 = t_2 \} \cup E'$, et

PDL = 


où
$\alpha_1$ repr $t_1$
$\alpha_2$ repr $t_2$
PDL' repr E'

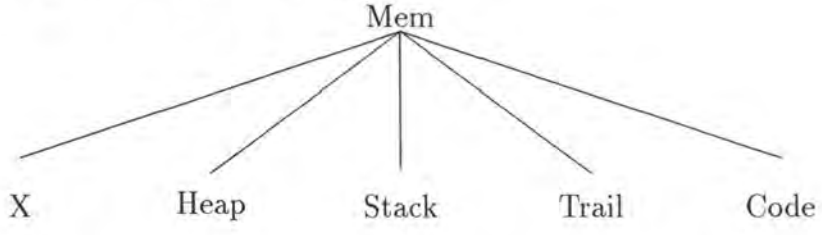
Clôturons ce chapitre en détaillant l'architecture de la mémoire WAM. Nous avons introduit précédemment diverses zones de travail telles que la pile de recopie ( Heap ), la pile de restauration ( Trail ), la pile des environnements ( Stack ) et les registres ( X ). A ces zones s'ajoutera encore l'aire d'encodage ( Code ) qui contiendra les instructions WAM générées. L'emplacement de telle ou telle zone en mémoire est, selon nous, tout à fait fortuit. Un découpage de la mémoire WAM se fera suivant les zones de données. La mémoire sera constituée d'abord des registres X puis du Heap et du Stack, suivi du Trail et finalement du Code.

La configuration de la mémoire est donc :

Mem :

| X |
|---|
| Heap |
| Stack |
| Trail |
| Code |

Comme les adresses $\alpha$ des termes $t_\alpha$ manipulées par les instructions doivent permettre d'accéder indifféremment à l'une ou l'autre zone, nous avons décidé de spécifier formellement la mémoire comme une union entre ces zones i.e.




Il nous revient donc de vérifier à tout instant qu'aucun registre global associé à une zone n'en sorte i.e. ne référence jamais une cellule mémoire qui n'appartient pas à sa zone.

Enonçons les associations et les contraintes qui en résultent :

| Registres | Zones | Contraintes |
|---|---|---|
| i | X | base_des_registres $\leq$ i < base_du_heap |
| H, HB | Heap | base_du_heap $\leq$ H, HB < base_du_stack |
| B, E | Stack | base_du_stack $\leq$ B, E < base_du_trail |
| TR | Trail | base_du_trail $\leq$ TR < base_du_code |
| P, CP | Code | base_du_code $\leq$ P, CP < mem_max |

( les registres B, E , P et CP seront expliqués ultérieurement. )

Faisons le point sur l'architecture de la mémoire. D'une part d'un point de vue théorique, la mémoire est une union de zones disjointes

$$\text{Mem} = \text{X } \dot{\cup}\text{Heap } \dot{\cup}\text{Stack } \dot{\cup}\text{Trail } \dot{\cup}\text{Code ;}$$

d'autre part d'un point de vue plus pratique, la mémoire est une union au sens de la spécification formelle, ce qui a pour conséquence de rendre équivalentes ces différentes zones

$$\text{Mem} = \text{X} = \text{Heap} = \text{Stack} = \text{Trail} = \text{Code}.$$

Illustrons cette conséquence. Soit $\alpha$ l'adresse d'une cellule de la mémoire. Peu importe la zone, on peut accéder à cette cellule par X[$\alpha$], Heap[$\alpha$], Stack[$\alpha$], Trail[$\alpha$], Code[$\alpha$]. Cette méthode d'accès n'a bien sûr de sens que pour la zone référencée par cette adresse. Ainsi, si $\alpha$ repr $t_\alpha$ où $t_\alpha$ est un terme de la pile de restauration, seul Trail[$\alpha$] a un sens.

Lors de l'implémentation, nous avons souhaité rester les plus proches possibles du pseudo-code des instructions. Donc, si dans le pseudo-code, une instruction réalise une affectation à une cellule du Heap telle que

$$\text{Heap[ H ]} \longleftarrow X_i$$

dans l'implémentation, nous établissons l'instruction

$$\text{Mem.Heap[ H ]} = \text{Mem.X [ i ]} ;$$

Nous avons choisi de représenter une cellule de la machine virtuelle par un mot de trente-deux bits. Ce mot se décompose différemment selon que nous sommes en présence d'une cellule de type | *tag* | *adresse* |, ou d'une cellule du type | *nom* | *arité* |. Pour la première cellule, nous avons codé le *tag* sur trois bits [1] et la partie *adresse* sur vingt-neuf bits. Quant à la seconde, elle est constituée de deux champs de seize bits. Le premier est un pointeur vers une table des noms et le second représente l'*arité*.

Les déclarations correspondantes sont :

1. au niveau des constantes :

```
#define mem_max constante \* taille de la memoire *\
```

2. au niveau des types :

```
typedef struct { unsigned nom ;
                 unsigned art ;
               } tcel ;

UNION { unsigned long mot ;
        tcel cel ;
      } tcellule ;

UNION { tcellule X[mem_max] ;
```

[1] Pour la machine non optimisée, seuls deux bits sont suffisants. Comme le signale Boizumault dans [Boi88], un minimum de trois bits est nécessaire pour la machine complète.

```
    tcellule Heap[mem_max] ;
    tcellule Stack[mem_max] ;
    tcellule Trail[mem_max] ;
    tcellule Code[mem_max] ;
  } tmem ;
```

3. au niveau des variables :

```
Mem : tmem ;
```

# Chapitre 3

# Construction de la machine abstraite de Warren

Comme nous l'avons signalé au chapitre précédent, WAM a fait le choix d'une représentation de termes par recopie et gère donc de manière différente les liaisons en construction et en décomposition. Cette représentation par recopie de structure induit un comportement différent de l'algorithme d'unification dans chacun des cas de liaisons. Elle privilégie l'accès aux instances de termes structurés. Par contre, la création de nouveaux termes à partir des squelettes nécessite la recopie de ces derniers. Bref, la recopie privilégie l'accès aux sous-termes. Elle est plus performante pour les programmes générant peu de nouveaux termes et travaillant en accès des termes existants. L'unification des variables libres entraîne la création au fur et à mesure d'une chaîne de liaison. La déréférenciation y est un procédé qui consiste à parcourir une telle chaîne de liaison afin d'en déterminer l'extremité. L'application systématique de ce procédé a pour effet de minimiser la taille de telles chaînes accélérant ainsi les accès ultérieurs aux valeurs des variables.

L'approche suggérée dans [AK91] comporte quatre étapes. Le but de cette approche est de considérer chaque caractéristique de la machine abstraite en introduisant pas à pas les différents aspects de Prolog. Ceci permet d'expliquer aussi simplement que possible les principes spécifiques propres à chaque aspect.

Lors de la première étape, nous ne considérons que l'unification de deux termes, opération principale en Prolog. Nous y introduisons les instructions spécifiques à cette opération et les structures des séquences d'instructions sous-jacentes. A la seconde étape, nous aborderons la résolution du problème composé d'une question à un but et d'un programme, ensemble de faits. Nous introduisons ainsi le premier niveau d'indexation des clauses

réalisé dans WAM. La troisième étape résout un problème similaire où, à présent, la question peut posséder plus d'un but et où le programme se compose de clauses dont le corps peut être non vide. Cette résolution se fait cependant sans backtracking. La dernière étape complète la machine en lui apportant le backtracking. Celui-ci peut être mis en place grâce au second niveau d'indexation des clauses réalisé dans WAM. Nous y gérons une pile d'environnements utile au backtracking. Rappellons que l'indexation des clauses a pour objectif de diminuer à chaque appel la taille du paquet de clauses envisageables évitant ainsi des tentatives inutiles d'unification et des créations inopinées de points de choix. L'apport de l'indexation se situe à deux niveaux. Un premier index est associé à chaque paquet de clauses déterminées par le prédicat de tête. Le second relie les différentes clauses d'un même paquet.

A chaque étape correspondra une machine $M_i$ et un langage $L_i$ . Toute machine $M_i$ disposera d'un certain nombre d'objets ( registres et ensembles de cellules ). Parmi ces objets, on retrouvera ceux de la machine précédente $M_{i-1}$ ainsi que de nouveaux qui constitueront la spécificité de la machine. Tout langage $L_i$ sera constitué d'un ensemble d'instructions. Ces instructions seront héritées du langage précédent $L_{i-1}$ moyennant certaines modifications correspondant à la machine $M_i$ ; ces instructions viendront s'ajouter aux nouvelles visant à compléter le langage. Dans la phase finale de construction, la machine $M_3$ et le langage $L_3$ correspondront à la machine pure abstraite de Warren. Nous appelons "pure" la machine qui n'a encore subi aucune optimisation.

Dans chaque section ultérieure consacrée à l'élaboration d'une machine, nous débuterons par une spécification des objets de la machine étudiée. Les objets spécifiés dans une machine resteront valables pour la machine suivante. Ensuite, nous apporterons une spécification formelle aux instructions du langage. Lors de celle-ci, nous ne ferons qu'énoncer les objets utilisés par celles-ci. Sans perte de généralité, nous supposerons que les objets non cités sont inchangés par les instructions. A ces spécifications seront associés les pseudo-codes decrits en grande partie dans [AK91]. Les séquences d'instructions sous-jacentes seront mises à jour et nous en démontrerons la correction.

## 3.1 W.A.M. $\langle L_0, M_0 \rangle$

Dans le langage $L_0$, seules deux entités peuvent être spécifiées : l'atome "programme" et l'atome "question". Ces deux atomes ne peuvent être des variables. La sémantique de ce langage est équivalente au calcul de l'unificateur le plus général entre ces deux termes. La portée des variables est limitée au terme dans lequel elles se situent. La signification d'un terme est donc bien naturellement indépendante du nom de ses variables.

L'idée de cette première machine est très simple.
Ayant un atome "programme" p défini, on lui soumet une question q et l'exécution soit échoue si p et q ne sont pas unifiables, soit réussit dans l'instanciation des termes de q obtenue par l'unification avec ceux de p.

La recopie des structures induisant un comportement différent de l'algorithme d'unification dans la liaison en construction et dans celle en décomposition, WAM introduit deux compilations distinctes pour les deux termes à unifier. La question est reconstituée sur le pile de recopie Heap par des instructions spécialisées " put_ " et " set_ " tandis que l'unification avec le terme programme est traitée par des instructions " get_ ". Chaque instruction " get_ " est complétée par un jeu d'instructions " unify_ " réalisant l'unification des sous-termes. Les instructions " unify_ " travaillent suivant deux modes Read et Write. Le mode Read réalise l'accès en décomposition d'un instance de terme et le mode Write correspond à un accès en construction ( nouvelle instance de terme ) et entraîne une recopie en sommet de pile.

En accord avec la sémantique opérationnelle de $L_0$, la compilation d'une question consiste à préparer d'abord les membres de droite d'un système d'équations à résoudre . Un terme sera traduit en une séquence d'instructions destinées à construire un exemplaire de ce terme dans le Heap à partir de sa forme textuelle.

Les registres, servant à stocker les données temporaires, sont alloués pour un terme sur base d'un index tel que (1) le registre $X_1$ soit toujours attribué au terme le plus extérieur et (2) le même registre soit attribué à toutes les occurences d'une variable donnée.
Ainsi, on alloue les registres de la façon suivante pour le terme p(Z,h(Z,W),f(W)) :

$$
\begin{aligned}
X_1 &= \mathrm{p}(X_1,X_2,X_3)\\
X_2 &= \mathrm{Z}\\
X_3 &= \mathrm{h}(X_2,X_5)\\
X_4 &= \mathrm{f}(X_5)\\
X_5 &= \mathrm{W}
\end{aligned}
$$

Cette mise en équations, lisant un terme de gauche à droite , est nommée "tokenization". Trois sortes d'éléments sont rencontrés :

- un registre associé au foncteur d'une structure ;
- un registre argument non encore rencontré précédemment lors de la lecture ;
- un registre argument déjà rencontré.

A cela correspondront trois instructions de la machine, respectivement :

- put_structure f/n $X_i$ ;
- set_variable $X_i$ ;
- set_value $X_i$ ;

où $X_i$ est le registre attribué.

Avec nos représentations du Heap, nous remarquons qu'il est utilisé comme une pile de termes construits. Nous conservons l'adresse de la première cellule libre du Heap dans le registre global H.

La compilation d'un terme "programme" se fait de manière similaire. Elle suppose cependant des instructions différentes car, si la compilation d'une question se faisait en construction, la compilation du programme réalisant l'unification avec la question peut s'effectuer soit en décomposition soit en construction. Cela dépendra des termes à unifier. Si le squelette du terme est déjà présent dans le Heap, la liaison se fait en décomposition sinon en construction. Une tokenization précède la traduction du terme en instructions. A nouveau, les instructions dépendent de ce qui a été rencontré :

- un registre associé au foncteur d'une structure ;
- un registre argument rencontré pour le première fois ;
- un registre argument déjà rencontré.

Les instructions deviennent alors, respectivement :

- get_structure f/n $X_i$ ;
- unify_variable $X_i$ ;
- unify_value $X_i$ ;

où $X_i$ est le registre attribué.

Dans la section Implémentation, nous présentons divers exemples pour lesquels nous avons généré les instructions de la machine. Notons encore que, dans le langage $L_0$, un échec interrompt toujours l'exécution en cours. Signalons également pour le mode READ ( décomposition ) la présence d'un pointeur S, registre global, indiquant à tout moment l'adresse, dans le Heap, du prochain terme à former.

### 3.1.1 Spécification et pseudo-code

**a. Spécification des objets utilisés**

**Heap :** pile de recopie glogale.

**H :** registre global indiquant la première cellule inoccupée du Heap.

$X_i$ : $i^e$ registre.

**STORE :** structure d'accès à toute la mémoire WAM.

**f/n :** construction de foncteur f et d'arité n.

$\alpha$ : adresse d'une cellule du Heap : base_du_heap $\leq \alpha < H$

**Deref :** adresse d'une cellule du Heap : base_du_heap $\leq \alpha < H$

$d_1, d_2$ : adresse d'une cellule du Heap : base_du_heap $\leq \alpha < H$

**mode :** variable signalant le mode de liaison ( décomposition READ ou construction WRITE )

**S :** registre du prochain terme à unifier.

**fail :** variable booléenne à vrai si deux termes ne sont pas unifiables, à faux sinon.

**t :** le terme à traiter.

**b. set_variable $X_i$**

- O.U. : Heap, H, $X_i$, t.
- Pré :

  $H_0 \geq$ base_du_heap
  $X_i$ est un registre inoccupé et i $\geq 1$.

- Post :

  $H = H_0 + 1$
  $\forall h$ base_du_heap $\leq h \leq H_0 - 1$ Heap[$h$] inchangé.
  Heap[$H_0$] = $\langle REF, H_0 \rangle$ d'où $H_0$ repr t.
  $X_i$ = Heap[$H_0$].

- Pseudo-code :

  Heap[H] $\longleftarrow$ $\langle$ REF , H $\rangle$
  $X_i \longleftarrow$ Heap[H]
  H $\longleftarrow$ H + 1

**c. set_value $X_i$**

- O.U. : Heap, H, $X_i$, t.
- Pré :

  $H_0 \geq$ base_du_heap
  $X_i$ est un registre avec i $\geq 1$ tel que
  $\exists h$ base_du_heap $\leq h \leq H_0 - 1$ $X_i$ = Heap[$h$] et $h$ repr t.

- Post :

  $H = H_0 + 1$
  $\forall h$ base_du_heap $\leq h \leq H_0 - 1$ Heap[$h$] inchangé.
  $X_i$ = Heap[$H_0$] d'où $H_0$ repr t.

- Pseudo-code :

Heap[H] $\longleftarrow X_i$
H $\longleftarrow$ H + 1

**d. put_structure f/n $X_i$**

- O.U. : Heap, H, $X_i$, t dont le foncteur et l'arité sont f/n.
- Pré :

  $H_0 \geq$ base_du_heap
  $X_i$ est un registre inoccupé et i $\geq 1$.
  $n \geq 0$

- Post :

  $H = H_0 + 2$
  $\forall h$ base_du_heap $\leq h \leq H_0 - 1$ Heap[$h$] inchangé.
  Heap[$H_0$] = $\langle STR, H_0 + 1 \rangle$.
  Heap[$H_0 + 1$] = f/n.
  $X_i$ = Heap[$H_0$].

- Pseudo-code :

  Heap[H] $\longleftarrow$ ⟨ STR , H+1 ⟩
  Heap[H+1] $\longleftarrow$ f/n
  $X_i \longleftarrow$ Heap[H]
  H $\longleftarrow$ H + 2

**e. La compilation d'une question**

Les trois instructions précédentes sont les instructions spécifiques à la compilation d'une question. Cette question, dans $M_0$ , doit être un terme qui n'est pas une variable. Ce terme est précédé de "?-".

On peut décrire la séquence d'instructions S ( t, i, { $X_1, ..., X_{l-1}$ } ) du langage $L_0$ générées par la compilation de la question de manière récursive où t est le terme à traiter, i est l'indice du registre associé au terme t, et $l$ est l'indice du premier registre libre i.e. $X_1, ..., X_{l-1}$ sont les registres utilisés.

Que vaut S ( t, i, { $X_1, ..., X_{l-1}$ } ) ?

Si t est une variable

Alors S est vide

Sinon S devient pour un terme de la forme f($u_1, ..., u_n$) :

S ( $u_1$, $j_1$, { $X_1, ..., X_{l_1-1}$ } )

...

S ( $u_n$, $j_n$, { $X_1, ..., X_{l_n-1}$ } )

put_structure f/n $X_i$

$SV_1$ $X_{j_1}$

...

$SV_n$ $X_{j_n}$

où

$SV_k$ = si $u_k$ est une nouvelle variable i.e. $\forall j < l_k$ not ($X_j$ repr $u_k$)
alors "set_variable"
sinon "set_value"

et $j_k$ est tel que $X_{j_k}$ repr $u_k$ où

$j_k$ = si $u_k$ est une variable déjà rencontrée i.e. $\exists j < l_k$ $X_j$ repr $u_k$
alors $j$
sinon $l_k$ (et $l_k = l_k + 1$).

La détermination des indices $l_k$ se fait en fonction de la méthode de tokenization. La méthode que nous utilisons se décrit comme suit : nous évaluons les indices des registres pour les arguments de la structure courante puis nous appliquons l'appel récursif sur le premier sous-terme de la structure. Une fois celui-ci complètement visité, nous passons au second sous-terme et ainsi de suite ( stratégie proche de celle en profondeur d'abord ).
Pour évaluer formellement ces $l_k$, nous introduisons les notations suivantes :

#a : le nombre de registres alloués aux arguments du terme courant tel que l'on alloue un registre à un argument uniquement s'il est une structure ou une variable libre ( $0 \leq$ # a $\leq$ arité de la structure ) ;

$\#u_k$ : le nombre de registres alloués lors de la tokenization du sous-terme $u_k$. Il est égal au nombre de variables libres ( première occurence ) ajouté au nombre de structures rencontrées.

Ainsi, pour p(f(W),h(g(X),Y),k(Z)) nous obtenons #a = 3, $\#u_1 = 2$ (f et W), $\#u_2 = 4$ (h, g, X et Y) et $\#u_3 = 2$ (k et Z) tandis que pour p(Z,h(Z,W),f(W)) nous avons #a = 3, $\#u_1 = 1$ (Z), $\#u_2 = 2$ (h et W) et $\#u_3 = 1$ (f) et pour p(Z,f(Z),Z), #a = 2, $\#u_1 = 1$ (Z), $\#u_2 = 1$ (f)et $\#u_3 = 0$.

Grâce à ces notations, nous pouvons définir précisement les indices $l_k$ selon les relations :

1. $k = 1 : l_1 = l + \#a$
2. $k > 1 : l_k = l_{k-1} + \#u_{k-1}$

Ces relations sont à mettre en lien avec les séquences vues précédemment
S( t, i, { $X_1$, ..., $X_l$ }) et S( $u_k$, $j_k$, { $X_1$, ..., $X_{l_k-1}$ })

Apportons une démonstration de la correction de cette séquence S( t, i, { $X_1$, ..., $X_l$ }) que nous venons d'introduire. Elle se fera par induction sur le terme traité à l'aide d'une relation bien fondée.

Quelques rappels sur cette notion :

1. Soit $\prec$ une relation binaire sur E,
   $\prec$ est **bien fondée**, si et seulement si, il n'existe pas de suite infinie $e_0, e_1, ..., e_i, ...$ dans E telle que ... $\prec e_{i+1} \prec e_i \prec ... \prec e_1 \prec e_0$.
   Note : si elle existe alors elle est finie.
2. Soit S $\subseteq$ E et soit e $\in$ S,
   e est **minimal** par rapport à $\prec$ dans S, si et seulement si, $\forall e' \in S : e' \not\prec e$
   Note : de plus, rien n'empêche d'avoir plusieurs minimaux.
3. Des deux définitions précédentes, on peut en déduire :
   $\prec$ est **bien fondée** dans E, si et seulement si, $\forall$ S $\subseteq$ E, S $\neq \emptyset$, S possède un élément **minimal**.

Dans notre cas, suivant la syntaxe Prolog établie au chapitre 1, nous pouvons énoncer la relation bien fondée suivante $\prec$ "est un sous-terme de".
En effet, si E représente l'ensemble des termes, alors les variables et les constantes sont des éléments minimaux de tout sous-ensemble de E. Par la propriété 3, la relation $\prec$ est ainsi bien fondée.

Soit la construction $t = f(t_1, ..., t_n)$, supposons par hypothèse que S soit correct pour les sous-termes $t_1$ à $t_n$ ie. $t_i \prec t \; \forall i$.
Montrons que cela l'est pour t : S ( t, i, { $X_1, ..., X_{l-1}$ } ).
Nous avons déjà la séquence intermédiaire d'instructions

S ( $t_1$, $j_1$, { $X_1, ..., X_{l_1-1}$ } )
...
S ( $t_n$, $j_n$, { $X_1, ..., X_{l_n-1}$ } )

A ce stade-ci, nous observons que Heap repr $(t_\alpha)_{\alpha \in I_0}$ où
$I_0 =$ { base_du_heap, ...,$H_0$-1} \ { adr( $\alpha$ ) : tag( $\alpha$ ) = STR et base_du_heap $\leq \alpha < H_0$}.
Une fois les sous-termes compilés, il suffit de générer la construction par son foncteur suivi de ses arguments. Nous pouvons donc écrire :

" put_structure f/n $X_i$ ".

Ensuite, nous avons la sous-séquence,

$$SV_k \ X_{j_k} \ \forall k \ 1 \leq k \leq n.$$

Soit le sous-terme $t_k$, deux situations sont possibles.

- $t_k$ est une nouvelle variable d'où S ( $t_k$, $j_k$, { $X_1, ..., X_{l_k-1}$ } ) = $\emptyset$ et dès lors $SV_k \ X_{j_k}$ devient " set_variable $X_{j_k}$ ".
- $t_k$ est soit une variable déjà rencontrée soit une construction d'où S ( $t_k$, $j_k$, { $X_1, ..., X_{l_k-1}$ } ) est correct car $t_k \prec t$ par hypothèse, et dès lors $SV_k \ X_{j_k}$ devient " set_value $X_{j_k}$ ".

La suite d'instructions ainsi générées correspond bien à la compilation du terme t. De cet ensemble d'instructions, on peut en déduire que

$$H_0 \text{ repr t et } H \geq H_0 + 1$$

D'où nous retrouvons la représentation du Heap :

$$\text{Heap repr } (t_\alpha)_{\alpha \in I}$$

où $I =$ { base_du_heap, ..., $H-1$}\{$adr(\alpha) : tag(\alpha) =$ STR et base_du_heap $\leq \alpha < H$}

### f. L'algorithme d'unification

Cet algorithme ne constitue pas en soi une instruction du langage $L_0$ mais l'instruction "unify_value" que nous verrons ci-après l'utilise.
De même, la fonction "Deref" et la procédure "Bind" que nous allons préalablement introduire sont toutes deux utilisées à la fois par l'algorithme et par l'instruction "get_structure".

1. **la fonction "Deref ( $\alpha$ : adresse)"**
   La déréférenciation est un procédé qui consiste à remonter la chaîne de liaison aboutissant à l'adresse $\alpha$ afin d'en déterminer l'adresse de l'extremité initiale.

   - O.U. : $\alpha$, t, Deref

- Pré : $\alpha$ repr t
- Post :

  Deref repr t' tel que t = t'$\sigma$ et
  $\exists h$ base_du_heap$\leq h \leq$ H-1 : $h$ repr t' et ( adr($h$) = $h$ ou tag($h$) = STR )

- Pseudo-code :

  $\langle$ tag, value $\rangle \longleftarrow$ Store [ $\alpha$ ]
  if (tag = REF) and (value $\neq \alpha$)
  then Deref $\longleftarrow$ Deref ( value )
  else Deref $\longleftarrow \alpha$

2. **la procédure "Bind($d_1$, $d_2$ : adresses)"**
   Le Bind établit la liaison entre une variable et un terme. Il est utile de signaler que la procédure "Bind" n'est nullement détaillée dans la machine $M_0$. Nous allons pallier à cette lacune en adaptant du mieux que possible le "Bind" fourni pour la machine complète optimisée dans [AK91].

   - O.U. : $d_1$, $d_2$, $t_1$, $t_2$, H
   - Pré :

     $d_1$ repr $t_1$
     $d_2$ repr $t_2$
     $t_1$ ou $t_2$ est une variable
     base_du_heap $\leq d_1$ , $d_2 \leq H_0$

   - Post :

     si ($t_1$ est une variable et $t_2$ une structure) ou ($t_1$ est une variable et base_du_heap $\leq d_2 < d_1 < H_0$)
     alors Store [ $d_1$ ] = Store$_0$ [ $d_2$ ] ie. $d_1$ repr $t_2$ et $d_2$ inchangé
     sinon Store [ $d_2$ ] = Store$_0$ [ $d_1$ ] ie. $d_2$ repr $t_1$ et $d_1$ inchangé

   - Pseudo-code :

     $\langle tag_1, - \rangle \longleftarrow$ Store [ $d_1$ ]
     $\langle tag_2, - \rangle \longleftarrow$ Store [ $d_2$ ]
     if ($tag_1$ = REf) and (($tag_2 \neq$ REF) or ($d_2 < d_1$))
     then Store [ $d_1$ ] $\longleftarrow$ Store [ $d_2$ ]
     else Store [ $d_2$ ] $\longleftarrow$ Store [ $d_1$ ]

3. **la fonction "Occur-Check ( $\alpha$, $\beta$ : adresse)"**
   cette fonction retourne vrai si la variable $t_\alpha$ est contenu dans le terme $t_\beta$, faux sinon.

- O.U. : $\alpha$, $\beta$, $t_\alpha$, $t_\beta$
- Pré :

  $\alpha$ repr $t_\alpha$ où $t_\alpha$ est une variable
  $\beta$ repr $t_\beta$ où $t_\beta$ est une terme quelconque
- Post :

  occur-check = ( $t_\alpha \in var(t_\beta)$ )
- Pseudo-code :

```
fail ⟵ false
d ⟵ Deref(β)
 if      tag(d) = REF
 then    if adr(d) = Deref(α)
         then fail ⟵ true
 else    i ⟵ 1
         while ( i ≤ arité(d+1) ) and ( not fail ) do
         begin
          fail ⟵ occur-check( α, adr(d)+i )
          i ⟵ i+1
         end
```

- Démontrons la correction de la fonction récursive que nous avons introduite.
  Considérons la relation bien fondée déjà introduite précédemment : $\prec$ "est un sous-terme de" dont, rappelons le, les variables et les constantes sont les éléments minimaux. Le paramètre d'induction sera le terme $t_\beta$.
  Soit le cas de base où $t_\beta$ est une variable, si Deref($\alpha$) = Deref($\beta$) alors vrai sinon faux.
  Soit le cas inductif où $t_\beta$ est une structure n-aire ( n $\geq$ 0 ),
  si n=0 alors faux
  sinon

  occur-check( $\alpha$, adr( $\beta$+1 ) ) $\wedge$
  ... $\wedge$
  occur-check( $\alpha$, adr($\beta$+n ) )

  Notons $\gamma_i$ = adr( $\beta$+1 ). Comme par hypothèse d'induction, nous supposons que l'occur-check est correct pour les sous-termes $t_{\gamma_i}$ et que $t_{\gamma_i} \prec t_\beta$, la correction est ainsi démontrée.

4. **l'algorithme Unify($\alpha_1$, $\alpha_2$ : adresses)**
   Cet algorithme réalise l'unification des deux termes $t_{\alpha_1}$ et $t_{\alpha_2}$.

- O.U. : Heap, H, $\alpha_1$, $\alpha_2$, $t_1$, $t_2$, fail.
- Pré :

  $\alpha_1$ repr $t_1$
  $\alpha_2$ repr $t_2$
  $H_0 \geq$ base_du_heap
  Heap$_0$ repr $(t_\alpha)_{\alpha \in I_0}$

- Post :

  Si $t_1$ et $t_2$ ne sont pas unifiables
  Alors fail (= true)
  sinon Heap repr $(t_\alpha \sigma)_{\alpha \in I}$
  où $\sigma$ est le mgu de l'unification de $t_1$ et $t_2$
  En particulier, $\alpha_1$ repr $t_1\sigma$ et $\alpha_2$ repr $t_2\sigma$

- Pseudo-code :

```
push(α1,PDL)
push(α2,PDL)
fail ⟵ false
while not (empty(PDL) or fail ) do
    d1 ⟵ Deref(pop(PDL))
    d2 ⟵ Deref(pop(PDL))
    if d1 ≠ d2 then
      ⟨t1, v1⟩ ⟵ Store[d1]
      ⟨t2, d2⟩ ⟵ Store[d2]
       if       (t1 = REF) or (t2 = REF) then
                Bind(d1,d2)
       else
       begin
                f1/n1 ⟵ Store[v1]
                f2/n2 ⟵ Store[v2]
                if (f1 = f2) and (n1 = n2) then
                for i ⟵ 1 to n1 do
                begin
                 push(v1 + i,PDL)
                 push(v2 + i,PDL)
                end
                else fail ⟵ true
       end
```

Etudions les propriétés de l'algorithme en tâchant d'y retrouver les règles de transition énoncées dans le premier chapitre. Reprenons ces règles une à une.

1. $\langle \sigma, \{\} \rangle \longmapsto \sigma$
   En effet, PDL repr E. Si PDL est vide alors PDL repr E={}. La condition d'arrêt est respecté et $\sigma = \sigma_0$.

2. $X \neq t \wedge X \notin Var(t)$
   $\langle \sigma, X = t :: E \rangle \longmapsto \langle \sigma\{X/t\}, E\{X/t\} \rangle$
   Dans Unify de la machine $M_0$, cette hypothèse n'est pas vérifiée. Dans tous les cas, après avoir retiré deux éléments de la pile PDL, on réalise la liaison par Bind si l'un des termes est une variable. En pratique, nous avons inséré un Occur-Check facultatif dans le Bind pour pallier à ce manque. Si l'Occur-Check est activé, il nous permet de vérifier que nous ne produirons pas de termes circulaires par une substitution du type X / f(X).

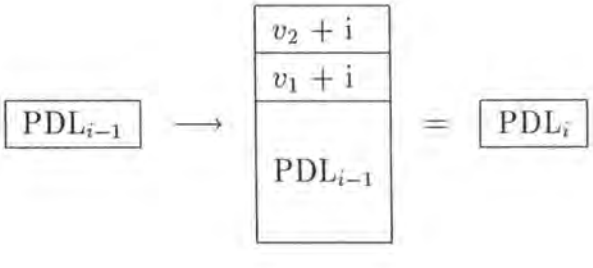
3. $X \neq t \wedge X \in Var(t)$
   $\langle \sigma, X = t :: E \rangle \longmapsto$ **fail**
   Cette règle était absente au départ. En insérant l'Occur-Check, l'unification échoue si la variable à instancier est présente dans l'autre terme.

4. $\langle \sigma, X = X :: E \rangle \longmapsto \langle \sigma, E \rangle$
   En effet, si $d_1$ repr $t_1$ et $d_2$ repr $t_2$ et $d_1 = d_2$ alors l'itération ne fait rien et $\sigma = \sigma_0$.

5. $\langle \sigma, t = X :: E \rangle \longmapsto \langle \sigma, X = t :: E \rangle$
   En effet, dans Bind,on substitue toujours la variable par le terme.

6. $(f \neq g) \vee (n \neq m)$
   $\langle \sigma, f(t_1, ..., t_n) = g(u_1, ..., u_m) :: E \rangle \longmapsto$ **fail**
   En effet, si f $\neq$ g ou n $\neq$ m alors l'algorithme produit fail et la condition d'arrêt est vérifiée.

7. $\langle \sigma, f(t_1, ..., t_n) = f(u_1, ..., u_n) :: E \rangle \longmapsto \langle \sigma, t_1 = u_1 :: ... :: t_n = u_n :: E \rangle$
   En effet, à l'itération i de la boucle FOR, on observe :



$PDL_0$ repr $E_0$ et $(v_2 + i)$ repr $t_2^i$ et $(v_1 + i)$ repr $t_1^i$ $\forall\ 1 \leq i \leq n$

$PDL_1$ repr $\{\ t_1^1 = t_2^1\ \} \cup E_0$ ;
...
$PDL_i$ repr $\{\ t_1^i = t_2^i\ \} \cup E_{i-1}$ ;
...
$PDL_n$ repr $\{\ t_1^n = t_2^n\ \} \cup E_{n-1}$ ;

Donc, $PDL_n$ repr $\{\ t_1^1 = t_2^1\ \} \cup ... \cup \{\ t_1^n = t_2^n\ \}$ $E_0$ ;
où $PDL_n$ est de la forme :

| $v_2 + n$ |
|---|
| $v_1 + n$ |
| ... |
| $v_2 + 1$ |
| $v_1 + 1$ |
| $PDL_0$ |

et $\sigma = \sigma_0$, la substitution est inchangée.

Quant à la terminaison de l'algorithme, elle est assurée par le nombre fini de variables que possède le premier terme. Chaque application du Bind lie l'une de ces variables. Quand chaque variable est liée, l'unification est réalisée et l'algorithme se termine.

**g. unify_variable** $X_i$

- O.U. : Heap, H, $X_i$, mode, S.
- Pré :

  $H_0$ > base_du_heap
  $X_i$ est le registre représentant le terme que l'on veut substituer par la variable.
  $S_0 \geq$ base_du_heap

- Post :

Si mode = Read
Alors

$X_i$ = Heap[S]
Heap inchangé
H = $H_0$

Sinon

Heap[$H_0$] = $\langle$ REF, $H_0\rangle$
Heap[h] inchangé $\forall$ h base_du_heap $\leq$ h < $H_0$
$X_i$ = Heap[$H_0$]
H = $H_0$ +1

S = $S_0$ +1

- Pseudo-code :

Case mode of
Read : $X_i \longleftarrow$ Heap[S]
Write : Heap[H] $\longleftarrow \langle$ REF, H $\rangle$
$X_i \longleftarrow$ Heap[H]
H $\longleftarrow$ H + 1
Endcase
S $\longleftarrow$ S + 1

**h. unify_value** $X_i$

- O.U. : Heap, H, $X_i$, mode, S.
- Pré :

$H_0$ > base_du_heap
$S_0 \geq$ base_du_heap
$X_i$ registre représentant le terme que l'on veut substituer par la variable.

- Post :

Si mode = Read
Alors $t_{\alpha_1}$ et $t_{\alpha_1}$ ont été unifiés où $\alpha_1 = X_i$ et $\alpha_2 = S_0$
Heap repr $(t_\alpha\sigma)_{\alpha\in I}$ où $\sigma$ = mgu($t_{\alpha_1}$,$t_{\alpha_2}$) S = $S_0$ + 1
Sinon Heap[H] = $X_i$
H = $H_0$ + 1

- Pseudo-code :

Case mode of

Read : Unify( $X_i$, S )

Write : Heap[H] $\longleftarrow$ $X_i$

H $\longleftarrow$ H + 1

Endcase

S $\longleftarrow$ S + 1

**i. get_structure f/n $X_i$**

- O.U. : Heap, H,f/n, $X_i$, mode, S, fail.
- Pré :

$H_0$ > base_du_heap

$S_0 \geq$ base_du_heap

n $\geq 0$

$X_i$ registre représentant le terme que l'on veut substituer par la structure.

- Post :

Si $\alpha$ = Deref($X_i$) et $t_\alpha$ est une variable

Alors

mode = Write

Heap[h] inchangé $\forall$ h base_du_heap $\leq$ h < $H_0$

Heap[$H_0$] = $\langle$ STR, $H_0$ + 1 $\rangle$

Heap[$H_0$ + 1] = f/n

H = $H_0$ +2

S = $S_0$ inchangé

fail = false

Sinon

Si les foncteurs sont identiques

Alors

$S_1$ = Deref($X_i$) + 1

mode = Read

Sinon fail = true

- Pseudo-code :

```
addr ⟵ Deref(X_i)
Case STORE(addr) of
    ⟨ REF, - ⟩ :    mode ⟵ Write
                    Heap[H] ⟵ ⟨ STR, H + 1 ⟩
                    Heap[H + 1] ⟵ f/n
                    Bind(addr,H)
                    H ⟵ H + 2
    ⟨ STR, a ⟩ :    if Heap[a] = f/n
                    then S ⟵ a + 1 ; mode ⟵ Read
                    else fail ⟵ true
    other :         fail ⟵ true
Endcase
```

### j. Compilation d'un programme

La machine $M_0$ n'admet comme programme qu'un terme écrit sur une seule ligne.
De façon similaire mais duale par rapport à la compilation d'une question, nous pouvons décrire la séquence d'instructions du langage $L_0$ générées par la compilation d'un programme.
Considérons à nouveau la séquence récursive S ( t, i, $\{X_1, ..., X_{l-1}\}$ où t est le terme à traiter, i est l'indice du registre associé au terme et l est l'indice du premier registre libre.

Que vaut S ( t, i, $\{X_1, ..., X_{l-1}\}$ ?
Si t est une variable
Alors S est vide
Sinon S est la séquence d'instructions suivante :

get_structure f/n $X_i$
$UV_1$ $X_{j_1}$
...
$UV_n$ $X_{j_n}$
S($u_1$,$j_1$,{ $X_1$, ..., $X_{l_1-1}$ })
...
S($u_n$,$j_n$,{ $X_1$, ..., $X_{l_n-1}$ })

où

$UV_k =$ si $u_k$ est une variable déjà rencontrée ie. $\exists j_k$ j < $l_k$ : $X_{j_k}$ repr $u_k$
alors unify_value $X_{j_k}$
sinon unify_variable $X_{j_k}$

et $j_k$ tel que $X_{j_k}$ repr $u_k$ où

$j_k =$ si $u_k$ est une variable déjà rencontrée ie. $\exists j_k$ j < $l_k$ : $X_{j_k}$ repr $u_k$
alors $j$
sinon $l_k$ (et $l_k = l_k + 1$)

et $l_k$ est défini comme pour une question c'est-à-dire,

1. $k = 1 : l_1 = l + \#a$
2. $k > 1 : l_k = l_{k-1} + \#u_{k-1}$

### 3.1.2 Exemples et implémentation des instructions

Exemples :

1. Soit le programme f(X,g(X,a)) et la question f(b,Y).
   Le code WAM généré est le suivant :

```
put_structure b/0 X2
put_structure f/2 X1
set_value X2
set_variable X3
get_structure f/2 X1
unify_variable X2
unify_variable X3
get_structure g/2 X3
unify_value X2
unify_variable X4
get_structure a/0 X4
```

   Solution(s) ? oui, f(b,g(b,a)).

2. Soit le programme p(f(a),g(X)) et la question p(Y,Y).
   Le code WAM généré est le suivant :

```
put_structure p/2 X1
set_variable X2
set_value X2
get_structure p/2 X1
unify_variable X2
unify_variable X3
get_structure f/1 X2
unify_variable X4
get_structure a/0 X4
get_structure g/2 X3
unify_variable X5
```

Solution(s) ? non, échec à l'unification.

3. Soit le programme p(X,g(X)) et la question p(Y,Y).
   Le code WAM généré est le suivant :

```
put_structure p/2 X1
set_variable X2
set_value X2
get_structure p/2 X1
unify_variable X2
unify_variable X3
get_structure g/1 X3
unify_value X2
```

   Solution(s) sans Occur-Check ? oui, p(g(g(g(...
   Solution(s) avec Occur-Check ? non, échec car occur-check .

4. Soit le programme p(f(X),h(Y,f(a)),Y) et la question p(Z,h(Z,W),f(W)).
   Le code WAM généré est le suivant :

```
put_structure h/2 X3
set_variable X2
set_variable X5
put_structure f/1 X4
set_value X5
put_structure p/3 X1
```

```
set_value X2
set_value X3
set_value X4
get_structure p/3 X1
unify_variable X2
unify_variable X3
unify_variable X4
get_structure f/1 X2
unify_variable X5
get_structure h/2 X3
unify_value X4
unify_variable X6
get_structure f/1 X6
unify_variable X7
get_structure a/0 X7
```

Solution(s) ? oui, p(f(f(a)),h(f(f(a)),f(a)),f(f(a)).

5. Soit le programme p(Z,h(Z,W),f(W)) et la question p(f(X),h(Y,f(a)),Y). Le code WAM généré est le suivant :

```
put_structure f/1 X2
set_value X5
put_structure a/0 X7
put_structure f/1 X6
set_value X7
put_structure h/2 X3
set_variable X4
set_value X6
put_structure p/3 X1
set_value X2
set_value X3
set_value X4
get_structure p/3 X1
unify_variable X2
unify_variable X3
unify_variable X4
get_structure h/2 X3
```

```
unify_value X2
unify_variable X6
get_structure f/1 X4
unify_value X6
```

Solution(s) ? oui, p(f(f(a)),h(f(f(a)),f(a)),f(f(a)).

## Implémentation :

```
void W0_put_structure(unsigned nom,unsigned arite,unsigned reg)
{
  tcellule cellule;
  cellule.cel.nom=nom;
  cellule.cel.art=arite;
  Mem.Heap[H].mot=(H+1)*8+1;
  Mem.Heap[H+1]=cellule;
  Mem.X[reg].mot=Mem.Heap[H].mot;
  H=H+2;
}

void W0_set_variable(unsigned reg)
{
  Mem.Heap[H].mot=H*8+0;
  Mem.X[reg].mot=Mem.Heap[H].mot;
  H++;
}

void W0_set_value(unsigned reg)
{
  Mem.Heap[H].mot=Mem.X[reg].mot;
  H++;
}

char W0_Unify(unsigned long a1,unsigned long  a2)
{
 tcellule cel1,cel2;
 unsigned long d1,d2,valeur,v1,v2,PDL[pile_max];
```

```
int echec,i,meme_nom;
unsigned spdl;
char t1,t2;

  spdl=0;
  Push(a1,&spdl,PDL);
  Push(a2,&spdl,PDL);
  echec=0;
  while (!(Empty(&spdl)||echec))
    {d1=Deref(Pop(&spdl,PDL));
     d2=Deref(Pop(&spdl,PDL));
     if (d1!=d2)
       {t1=type(Mem.Heap[d1].mot);
        t2=type(Mem.Heap[d2].mot);
        v1=adresse(Mem.Heap[d1].mot);
        v2=adresse(Mem.Heap[d2].mot);
        if ((t1==0) || (t2==0))
          WO_Bind(d1,d2);
        else
        {
         cel1=Mem.Heap[v1];
         cel2=Mem.Heap[v2];
         meme_nom=strcmp(tabfonct[cel1.cel.nom].nom,tabfonct[cel2.cel.nom].nom);
         if ((meme_nom == 0)&&(cel1.cel.art == cel2.cel.art))
         {for(i=1;i<=cel1.cel.art;i++)
          {Push(v1+i,&spdl,PDL);
           Push(v2+i,&spdl,PDL);}
         }
  else
           echec=1;
        };
};
    };
 return(echec) ;
}

char WO_get_structure(unsigned nom,unsigned arite,unsigned reg)
{
```

```
  char echec;
  unsigned long a,adr;
  tcellule cellule;

  echec=0;
  adr=Deref(reg);
  switch(type(Mem.Store[adr]))
  { case 0 : {
       Mem.Heap[H].mot=(H+1)*8+1;
       cellule.cel.nom=nom;
       cellule.cel.art=arite;
       Mem.Heap[H+1]=cellule;
       W0_Bind(adr,H);
       mode='w';
       H=H+2;
       break;
     }
    case 1 : {
       a=adresse(Mem.Store[adr]);
       cellule=Mem.Heap[a];
       if ((strcmp(tabfonct[cellule.cel.nom].nom,tabfonct[nom].nom)==0)
       &&(cellule.cel.art)==arite)
       { S=a+1;
                 mode='r';
       }
       else
         echec=1;
       break;
     }
    default : echec=1;
  }
  return(echec);
}

void W0_unify_variable(unsigned reg)
{
  switch(mode)
  { case 'r' : Mem.X[reg].mot=Mem.Heap[S].mot;break;
```

```
    case 'w' : {
                  Mem.Heap[H].mot=(H*8)+0;
                  Mem.X[reg].mot=Mem.Heap[H].mot;
                  H++;
                  break;
                }
  }
  S++;
}

char W0_unify_value(unsigned reg)
{
 char echec ;

 echec=0;
 switch(mode)
  { case 'r' : {
                echec = W0_Unify(reg,S);
                break;
       }
    case 'w' : {
                  Mem.Heap[H].mot=Mem.X[reg].mot ;
 echec=occur_check(H,Deref(H));
                  if (echec == 1)
                  { puts("! occur check !");
                    exit(1);
                  }
                  H++;
                  break;
                }
  }
  S++;
  return(echec);
}

char occur_check(unsigned long a1,unsigned long a2)
{
  int i;
```

```
  unsigned arite;
  unsigned long l;
  char echec;

  echec=0;
  i=1;
  if (type(Mem.Heap[a2].mot) != 0)
  { l=adresse(Mem.Heap[a2].mot);
    arite=Mem.Heap[l].cel.art;
    while ((i != arite+1)&&(echec == 0))
    { if (type(Mem.Heap[l+i].mot) == 0)
      { if (adresse(Mem.Heap[l+i].mot) == adresse(Mem.Heap[a1].mot))
          echec=1;
      }
      else
        echec=occur_check(a1,l+i);
      i++;
    }
  }
  return(echec);
}

unsigned long Deref(unsigned long adr)
{
  unsigned long cel;

  cel=Mem.Store[adr];
  if ((type(cel)==0)&&(adresse(cel)!=adr))
    return(Deref(adresse(cel)));
  else
    return(adr);
}

void W0_Bind(unsigned long a1,unsigned long a2)
{
 char echec,t1,t2;

 t1 = type ( Mem.Store[a1] );
```

```
t2 = type ( Mem.Store[a2] );
if ((t1==0)&&((t2!=0)||(a2<a1)))
{ if (mode == 'r')
  { echec=occur_check(a1,a2);
    if (echec != 0)
    { puts("! occur check !");
      exit(1);
    }
  }
  Mem.Store[a1]=Mem.Store[a2];
}
else
{ if (mode == 'r')
  { echec=occur_check(a2,a1);
    if (echec != 0)
    { puts("! occur check !");
      exit(1);
    }
  }
  Mem.Store[a2]=Mem.Store[a1];
}
}
```

## 3.2 W.A.M. $\langle L_1, M_1 \rangle$

Dans la machine précédente, nous avons introduit l'unification de deux atomes où chacun est une structure. A présent, nous allons nous intéresser aux arguments d'un prédicat. Le langage $L_1$ reste proche du langage $L_0$. Un programme y devient un ensemble de faits où chaque fait est identifié par le nom du prédicat. L'exécution d'une question produit une unification avec la définition appropriée ou échoue si aucun prédicat n'y correspond. Cette résolution apporte le premier niveau d'indexation des clauses.

Les instructions de la machine $M_0$ restent valables dans la machine $M_1$. Nous y introduisons un nouveau tableau CODE qui contiendra les instructions WAM générées. Deux instructions de contrôle sont proposées afin de permettre des sauts inconditionnels lors de l'exécution. Il s'agit du "call p/n", saut inconditionnel à l'adresse du début de la séquence d'instructions pour le fait dont le nom du prédicat est p/n, et "proceed", instruction de fin de séquence d'instruction pour un fait.

L'évolution principale de la machine consiste en l'élimination des symboles prédicats du Heap. Le problème d'unification devient la résolution de l'unification de plusieurs couples de termes simultanément. Dans cette optique, une organisation des registres est suggérée.

Nous prendrons dorénavant la convention suivante. Le registre $A_i$ est utilisé pour représenter l'argument d'un littéral, sinon le registre $X_i$ est conservé. Précisons que les registres $A_i$ et $X_i$ forment une partition de l'ensemble de registres.
Par exemple, considérons le littéral p(Z,h(Z,W),f(W)). Les registres sont alloués pour les termes de ce littéral de la façon suivante :

$$\begin{aligned} A_1 &= \text{Z}. \\ A_2 &= \text{h}(\ A_1,\ X_4\ ). \\ A_3 &= \text{f}(\ X_4\ ). \\ X_4 &= \text{W}. \end{aligned}$$

Pour manipuler correctement ces registres et plus précisement les arguments des prédicats, diverses instructions prendront en compte s'il s'agit de la première occurrence d'une variable argument ou non soit dans une question soit dans un fait.
Pour une question, la première occurrence d'une variable dans le $i^e$ argument entraîne la construction d'une nouvelle cellule REF sur le Heap et la copie de la variable dans le registre $A_i$ ; les autres occurrences induisent la copie de leur valeur dans le registre $A_i$.
Pour un fait, la première occurrence d'une variable dans le $i^e$ argument pose sa valeur dans

le registre $A_i$ ; les autres occurrences s'unifient avec la valeur du registre $A_i$.

Les instructions correspondantes sont :

1. put_variable $X_n$, $A_i$
2. put_value $X_n$, $A_i$
3. get_variable $X_n$, $A_i$
4. get_value $X_n$, $A_i$

### 3.2.1 Spécification et pseudo-code

Les instructions "set_" et "unify_"de la machine $M_0$ restent valables et correctement spécifiées. En fait, elles ne concernent pas les arguments mais uniquement leurs sous-termes.

#### a. Spécification des objets utilisés

Les objets sont ceux de la machine précédente auxquels nous ajouteront les nouveaux registres, de plus nous distinguerons le terme programme et le terme question.

$A_i$ : le registre associé au $i^e$ argument ;

**u** : le terme associé au fait ;

**t** : le terme associé à la question.

#### b. put_variable $X_n$, $A_i$

- O.U. : Heap, H, $X_n$, $A_i$, t.
- Pré :

La variable t est rencontrée pour la première fois en $i^e$ position dans les arguments de la question.
$X_n$ est le premier registre libre d'indice n $\geq 1$
$A_i$ est le registre associé au $i^e$ argument
Heap$_0$ repr $(t_\alpha)_{\alpha \in I_0}$ où
$I_0$ = { base_du_heap, ..., $H_0 - 1$ } \ { adr($\alpha$) : tag($\alpha$) = STR et base_du_heap $\leq \alpha \leq H_0 - 1$}
$H_0 \geq 0$

- Post :

Heap[$H_0$] = $\langle$ REF, $H_0$ $\rangle$ d'où $H_0$ repr t
$X_n$ = Heap[$H_0$]
$A_i$ = Heap[$H_0$]
H = $H_0 + 1$
Heap repr $(t_\alpha)_{\alpha \in I}$ où
$I$ = { base_du_heap, ..., H-1 } \ { adr($\alpha$) : tag($\alpha$) = STR et base_du_heap $\leq \alpha \leq H - 1$}

- Pseudo-code :

Heap[ H ] $\longleftarrow$ $\langle$ REF, H $\rangle$
$X_n \longleftarrow$ Heap[ H ]
$A_i \longleftarrow$ Heap[ H ]
H $\longleftarrow$ H + 1

**c. put_value** $X_n$, $A_i$

- O.U. : $X_n$, $A_i$, t.

- Pré :

La variable t a déjà été rencontrée :
$\exists\ h$, base_du_heap $\leq h \leq$ H-1, $h$ repr t
et $\exists\ n$, $1 \leq n <$ *libre*, $X_n$ repr t
où *libre* est l'indice du premier registre libre
$A_i$ est le registre associé au $i^e$ argument

- Post :

  $A_i = X_n$ d'où $A_i$ repr t
  $X_n$ inchangé

- Pseudo-code :

  $A_i \longleftarrow X_n$

**d. put_structure f/n, $A_i$**

L'instruction de la machine précédente reste correctement spécifiée si ce n'est la substitution de $X_i$ par $A_i$. Signalons qu'il y a bien deux instructions à présent l'une avec $X_i$ pour les sous-termes et l'autre avec $A_i$ pour les arguments. Ce n'est donc pas l'ancienne instruction transformée en nouvelle.

**e. call p/n**

- O.U. : p/n, P.

- Pré :

  p est le nom du prédicat de la question et n son arité.
  $P_0$ indique l'instruction courante, en l'occurrence "call p/n".

- Post :

  S'il existe un fait de nom de prédicat p/n
  Alors P indique le début de la séquence liée à ce fait
  Sinon P = 0

- Pseudo-code :

  P $\longleftarrow$ @(p/n) où @(p/n) signifie l'adresse du début de la séquence d'instructions liées au prédicat p/n.

### f. la compilation d'une question

Notons $S_0$ la séquence d'instructions générées dans le langage $L_0$. Nous appliquerons à présent une nouvelle séquence $S_1$ d'instructions de $L_1$ à chaque argument de la question. Nous clôturerons la compilation par un appel au fait qui a le même nom de prédicat que la question.
Ainsi pour une question $f(u_1, ..., u_n)$, la compilation produira la séquence suivante :

$S_1(u_1)$
...
$S_1(u_n)$
call f/n

où chaque $S_1(u_i)$ est une séquence faisant référence à la séquence $S_0$ de la machine précédente.

$S_1(u_i)$ devient :
Si $u_i$ est une variable

soit en première occurrence et n le premier registre libre :

"put_variable $X_n$ , $A_i$"

soit déjà rencontrée en $X_n$ :

"put_value $X_n$ , $A_i$"

Sinon $u_i$ est une structure du type $p(t_1, ..., t_m)$ :

$S_0$ ( $t_1$, $j_1$, { $X_1, ..., X_{l_1-1}$ } )
...
$S_0$ ( $t_m$, $j_m$, { $X_1, ..., X_{l_m-1}$ } )
put_structure p/m $A_i$
$SV_1$ $X_{j_1}$
...
$SV_m$ $X_{j_m}$

où les indices $j_k$ et $l_k$ restent définis comme précédemment. La définition récursive restant liée à $S_0$, nous n'en redémontrerons pas la correction.

**g. get_variable** $X_n$, $A_i$

- O.U. : $X_n$, $A_i$, u.
- Pré :

  la variable u est rencontrée pour la première fois en $i^e$ argument du fait.
  $X_n$ est le premier registre libre
  $A_i$ est le registre associé au $i^e$ argument tel que $A_i$ repr u

- Post :

  $X_n = A_i$ d'où $X_n$ repr u
  $A_i$ inchangé

- Pseudo-code :

  $X_n \longleftarrow A_i$

**h. get_value** $X_n$, $A_i$

- O.U. : Heap, $X_n$, $A_i$, u, t.
- Pré :

  la variable t a déjà été rencontrée : $\exists\ n,\ 1 \leq n < libre$, $X_n$ repr t
  $A_i$ est le registre associé au $i^e$ argument tel que $A_i$ repr u
  $\text{Heap}_0$ repr $(t_\alpha)_{\alpha \in I}$ où
  $I$ = { base_du_heap, ..., $H-1$ } \ { adr($\alpha$) : tag($\alpha$) = STR et
  base_du_heap $\leq \alpha \leq H-1$}

- Post :

  Si t et u ne sont pas unifiables,
  Alors fail = true
  Sinon

  Heap repr $(t_\alpha \sigma)_{\alpha \in I}$ où
  $I$ = { base_du_heap, ..., $H-1$ } \ { adr($\alpha$) : tag($\alpha$) = STR et
  base_du_heap $\leq \alpha \leq H-1$}
  et $\sigma$ est le mgu ( t, u ).

$X_n$ repr t$\sigma$
$A_i$ repr u$\sigma$

- Pseudo-code :

  Unify( $X_n$ , $A_i$ )

- Note : la substitution porte sur tous les termes construits sur le Heap et non seulement sur les termes u et t.

### i. get_structure f/n, $A_i$

Comme pour le "put_structure", l'instruction de $L_0$ reste correctement spécifiée moyennant substitution de $X_i$ par $A_i$. Nous obtenons à nouveau deux instructions, l'une avec $X_i$ pour les sou-termes et l'autre avec $A_i$ pour les arguments.

### j. proceed

Cette instruction n'entraîne pour l'instant aucune modification de nos états. Sa seule utilité actuelle est de signaler la fin d'une séquence d'instructions liées à un fait nommé.

### k. la compilation d'un fait

Soit la séquence $S_0$ d'instructions générées pour la compilation d'un programme dans $L_0$. Appliquons une séquence $S_1$ d'instructions de $L_1$ à chaque argument d'un fait et clôturons la compilation de ce fait par une instruction "proceed".
Ainsi pour un fait f( $u_1$ , ... , $u_n$ ), la compilation produira la séquence suivante :

f/n : $S_1(u_1)$
...
$S_1(u_n)$
proceed

où chaque $S_1(u_i)$ est une séquence se référant à la séquence $S_0$ définie dans la machine précédente et f/n est une étiquette indiquant la première instruction du fait.

$S_1(u_i)$ devient :
Si $u_i$ est une variable

soit en première occurrence et n le premier registre libre :

"get_variable $X_n$ , $A_i$"

soit déjà rencontrée en $X_n$ :

"get_value $X_n$ , $A_i$"

Sinon $u_i$ est une structure du type $p(t_1, ..., t_m)$ :

get_structure p/m $A_i$
$UV_1$ $X_{j_1}$
...
$UV_m$ $X_{j_m}$
$S_0$ ( $t_1$, $j_1$, { $X_1, ..., X_{l_1-1}$ } )
...
$S_0$ ( $t_m$, $j_m$, { $X_1, ..., X_{l_m-1}$ } )

### l. note sur le CODE

Chaque instruction est à présent codée et stockée dans une table notée Code. Nous avons décidé d'associer à chaque instruction un entier l'identifiant ainsi qu'une taille correspondant au nombre de mots qu'elle occupera dans le Code. Ces deux informations seront stockées sur un mot de la mémoire WAM. Les mots suivants contiendront chacun deux opérandes. Nous considérons comme opérandes l'indice des registres des instructions, le nom du foncteur et son arité. Un registre global P contiendra l'adresse de l'instruction courante.
Ainsi par exemple, pour l'instruction "get_structure f/n $A_i$"identifiée par l'entier 5 et de taille 3, nous observerons dans le Code :

| | |
|---|---|
| ... | ... |
| 5 | 3 |
| f | n |
| i | $\emptyset$ |
| ... | ... |

Signalons qu'en réalité, nous ne stockerons pas la chaîne de caractères du foncteur dans le Code mais son indice dans la table des foncteurs. Pour plus de détails, nous renvoyons le lecteur au chapitre consacré à l'analyseur de termes.

Il serait prématuré de fournir dès à présent la liste des affectations des entiers aux instructions pour cette machine. D'autres instructions viendront encore s'ajouter et modifieront la liste. Toutefois, il est possible au lecteur de satisfaire sa curiosité en se plongeant dans le code source de l'analyseur du langage $L_1$ à la fonction "execution()".

### 3.2.2 Exemples et implémentation des instructions

Exemples :

1. Soit le programme composé des faits suivant :
   g(X,h(T,f(X))).
   p(Z,h(Z,W),f(W)).
   Le code WAM du programme est le suivant :

```
g/2 : get_variable X5,A1
      get_structure h/2 A2
      unify_variable X3
      unify_variable X4
      get_structure f/1 X4
      unify_value X5
      proceed
p/3 : get_variable X4,A1
      get_structure h/2 A2
      unify_value X4
      unify_variable X5
      get_structure f/1 X3
      unify_value X5
      proceed
```

   et la question p(f(X),h(Y,f(a)),Y).
   Le code WAM de la question est le suivant :

```
put_structure f/1 A1
set_variable X4
put_structure a/0 X7
put_structure f/1 X6
set_value X7
put_structure h/2 A2
set_variable X5
set_value X6
put_value X5,A3
call p/3
```

Solution(s) ? oui, p(f(f(a)),h(f(f(a)),f(a)),f(f(a))).

2. Soit le programme avec les mêmes faits et la question g(Y,Y).
   Le code WAM de la question est le suivant :

```
put_variable X3,A1
put_value X3,A2
call g/2
```

Solution(s) sans Occur-Check ? oui, g(h(Var_24,f(h(Var_24,f(h(Var_24,...
Solution(s) avec Occur-Check ? non, échec avec occur-check.

## Implémentation :

```
void W1_put_variable(unsigned regx,unsigned rega)
{
  Mem.Heap[H].mot=H*8+0;
  Mem.X[regx].mot=Mem.Heap[H].mot;
  Mem.X[rega].mot=Mem.Heap[H].mot;
  H++;
}

void W1_put_value(unsigned regx,unsigned rega)
{
  Mem.X[rega].mot=Mem.X[regx].mot;
```

```
}

void W1_get_variable(unsigned regx,unsigned rega)
{
  Mem.X[regx].mot=Mem.X[rega].mot;
}

char W1_get_value(unsigned regx,unsigned rega)
{
  return(W1_Unify(regx,rega));
}

unsigned long W1_call( unsigned nom)
{
  return(nom);
}

unsigned W1_proceed ( unsigned fin )
{
 return( fin ) ;
}
```

## 3.3 W.A.M. $\langle L_2, M_2 \rangle$

La nouvelle extension du langage $L_2$ considère à présent les procédures non seulement comme des faits mais aussi comme des clauses dont le corps peut être non vide.
Un programme y est défini comme un ensemble de procédures, au plus une clause par nom de prédicat, de la forme "$a_0$ :- $a_1$ , ... , $a_n$ ." où n $\geq$ 0. Quand n = 0, la clause est un fait de la forme "$a_0$ .". Si n > 0, la clause a une tête $a_0$ et un corps de buts $a_1$ , ... , $a_n$ .
Une question devient une séquence de buts telle que "?- $g_1$ , ... , $g_k$ ." où k > 0.
Précisons que, comme en Prolog pur, la portée des variables est limitée à la clause ou à la question dans laquelle elles apparaissent.

Dans ce contexte, l'unification se fait par technique de résolution gauche-droite sur la question. On unifie chaque but $g_i$ avec la tête de la clause correspondante ( si elle existe sinon il y a échec ). Si l'unification se déroule avec succès, on tient compte de la portée des variables liées par l'unification. Donc, l'exécution soit se termine avec succès, soit échoue, soit ne se termine jamais ( en effet rien n'empêche la présence de branches infinies dans l'arbre de recherche ). Ce langage n'admettant qu'une clause par nom de prédicat, l'absence de points de choix exclut la notion de backtracking dans ce langage.

Toutes les instructions du langage $L_1$ peuvent être utilisées pour les clauses du nouveau langage $L_2$. Deux précautions doivent cependant être prises : l'une concernant la continuation de l'exécution d'une séquence d'instructions d'un but, l'autre visant à résoudre les conflits possibles lors de l'utilisation des registres.

Les faits peuvent être compilés suivant le même schéma que la machine précédente. Seules les instructions de contrôle seront modifiées afin de permettre la reprise de l'exécution après un appel. A cette occasion, un nouveau registre global CP est introduit. Il contiendra l'adresse de la prochaine instruction à exécuter après le retour d'un appel avec succès. Les instructions de contrôle veilleront à le mettre à jour et à le restaurer correctement.

Même si une clause généralise un fait dans le sens où un fait est une clause sans corps, dans la machine abstraite de Warren, la compilation du corps d'une clause généralise celle d'une question. En effet, une question peut être perçue comme une clause sans tête. L'idée sous-jacente à la compilation d'une clause est d'utiliser les instructions du langage $L_1$ en manipulant d'une part la tête de la clause comme un fait de $L_1$ i.e. un terme programme et d'autre part chaque but du corps comme une question de $L_1$. Ainsi pour la clause

$p_0(...)$ :- $p_1(...)$, ... , $p_n(...)$ ,la séquence d'instructions générées sera de la forme :

compilation d'un atome programme $p_0$ : instructions get_...
compilation d'un atome question $p_1$ : instructions put_...
...
compilation d'un atome question $p_n$ : instructions put_...

Chaque compilation d'un atome question $p_i$ se termine par l'instruction de contrôle "call $p_i$".

Voyons ce qu'il en est de l'utilisation des registres. Lors de l'appel d'un but, chaque argument d'un atome est chargé dans le registre $A_i$ correspondant. WAM accorde un statut particulier aux variables figurant dans un seul but. En effet, puisqu'aucun autre but n'utilisera leur valeur, nous pouvons songer à traiter directement de telles variables à l'aide des registres $X_i$. Elles ne feront l'objet d'aucune allocation dans une zone de sauvegarde. Selon [AK91], pour définir la permanence d'une variable, la tête de la clause est considérée comme une partie du premier but s'il existe. Une variable est dite temporaire si elle n'apparaît que dans un but. Elle est, sinon, permanente. Nous distinguerons dès lors les variables permanentes par des registres $Y_i$ dont la portée est supérieure à un but et les variables temporaires par les registres $X_i$. Ainsi pour p(X,Y) :- q(X,Z), r(Z,Y) les variables Z et Y doivent rester accessibles sur plus d'un but ( atomes p et r pour Y, atomes q et r pour Z ). Par contre, X est temporaire. En effet, p et q ne formant qu'un seul but, X n'apparaît qu'une fois. Il n'est donc pas nécessaire de le sauvegarder.

Les informations à restaurer seront sauvées dans la pile d'environnements Stack. L'adresse du début de l'environnement courant est conservée dans le registre global E. La sauvegarde de ces informations nécessite deux nouvelles instructions encadrant le code généré pour une clause non fait ou une question. Nous définissons alors "allocate" et "deallocate" dont les effets sont respectivement :

- d'allouer un environnement dans le Stack, contenant entre autres CE le pointeur de l'environnement précédent et le point de continuation de l'exécution CP ; et
- de supprimer l'environnement situé au point E, restaurant P à la valeur de CP et E à celle de CE.

Détaillons le contenu d'un environnement du Stack. Sont stockés, tout d'abord, l'adresse CP de la prochaine instruction à exécuter dans le Code, ensuite l'adresse CE de l'environnement précédent dans le Stack, le nombre N de variables permanentes ( permettant de déterminer la taille de l'environnement i.e. N+3 ) où $N \geq 0$, et les N variables permanentes

( cellule du premier type ). Comme il est possible de ne pas avoir de variables permanentes ( N = 0 ), nous admettons des environnements sans variables permanentes. L'élimination de ce cas particulier appartient au domaine de l'optimisation que nous n'aborderons pas dans ce travail.
La structure d'un environnement sera donc :

| | | |
|---|---|---|
| E : | CE | ( point de l'environnement précédent ) |
| E+1 : | CP | ( point de continuation de l'exécution ) |
| E+2 : | N | ( nombre de variables permanentes ) |
| E+3 : | $Y_1$ | ( $1^e$ variable permanente ) |
| ... | ... | |
| E+N+2 : | $Y_N$ | ( $N^e$ variable permanente ) |

### 3.3.1 Spécification et pseudo-code

Les instructions de la machine précédente sont pratiquement identiques excepté qu'à présent elles doivent permettre d'accéder soit aux variables temporaires soit aux variables permanentes. L'accès en mémoire WAM étant différent, les registres X pour les temporaires et le Stack pour les permanentes, nous avons préféré dédoubler les instructions en les indicant d'un "_temporary" ou d'un "_permanent" suivant le cas.

Les spécifications des instructions des variables restent correctes si l'on substitue $X_i$ par $Y_i$ i.e. Stack [E+i+2]. Dès lors, nous ne présenterons ici qu'un bref exposé des pseudo-codes mis à jour.

#### a. Spécification des objets utilisés

**Stack** : pile des environnements sauvegardés pour les clauses non faits et la question.

$Y_i$ : registre alloué dans le Stack pour la $i^e$ variable permanente.

**E** : pointeur vers le début de l'environnement courant dans le Stack.

**N** : nombre de variables permanentes à réserver.

Abordons les deux nouvelles instructions de contrôle du langage $L_2$.

**b. Allocate N**

- O.U. : N, CP, E, Stack.
- Pré :

  $N \geq 0$
  $E_0$ pointe vers le début de l'environnement courant
  CP est le point de continuation de l'exécution
  $Stack_0$ est l'ensemble des environnements alloués

- Post :

  E pointe vers le nouvel environnement courant :
  E = $E_0$ + Stack[ $E_0$+2 ] + 3
  Stack = $Stack_0$ ∪ le nouvel environnement tel que :
  Stack[ E ] = $E_0$
  Stack[ E+1 ] = CP
  Stack[ E+2 ] = N
  Les N cellules suivantes sont réservées pour les variables permanentes

- Pseudo-code :

  newE ⟵ E + Stack[ E+2 ] + 3
  Stack[ E ] ⟵ E
  Stack[ E ] ⟵ CP
  Stack[ E ] ⟵ N
  E ⟵ newE

**c. Deallocate**

- O.U. : P, E, Stack.
- Pré :

  $E_0$ pointe vers le début de l'environnement courant à supprimer
  $P_0$ pointe vers l'instruction Deallocate dans le CODE
  $Stack_0$ est l'ensemble des environnements alloués

- Post :

E pointe vers l'environnement précédent : E = Stack[ $E_0$ ]
P est le point de continuation de l'exécution : P = CP = Stack[ $E_0$+1 ]
Stack = $\text{Stack}_0$ \ l'environnement pointé par E
Stack[i] = $\text{Stack[i]}_0$ inchangé $\forall$ base_du_heap $\leq$ i < $E_0$

- Pseudo-code :

P ⟵ Stack[ E+1 ]
E ⟵ Stack[ E ]

### d. Compilation d'une clause

Notons Sql la séquence d'instructions générées pour la compilation d'un atome question dans $\langle L_1, M_1 \rangle$ et Spl la séquence d'instructions générées pour la compilation d'un fait (sans le proceed).
Soit la clause $p_0( u_1^0, ..., u_{n_0}^0 )$ :- $p_1( u_1^1, ..., u_{n_1}^1 ), ..., p_m( u_1^m, ..., u_{n_m}^m )$ où $n_k$ est l'arité du prédicat $p_k$ et $u_j^k$ est le $j^e$ argument du prédicat $p_k$.
La séquence d'instructions générées pour une telle clause sera :

si m > 0 : Allocate N
$\text{Spl}( p_0( u_1^0, ..., u_{n_0}^0 ) )$
$\text{Sql}( p_1( u_1^1, ..., u_{n_1}^1 ) )$
dont la dernière instruction est "call $p_1/n_1$
...
$\text{Sql}( p_m( u_1^m, ..., u_{n_m}^m ) )$
dont la dernière instruction est "call $p_m/n_m$
Deallocate

si m = 0 : $\text{Spl}( p_0( u_1^0, ..., u_{n_0}^0 ) )$
proceed

où N est le nombre de variables permanentes de la clause.

### e. Compilation d'une question

En conservant les notations de la compilation d'une clause, nous pouvons particulariser la séquence pour une question.

Soit la question ?-$p_1(\ u_1^1,\ ...,\ u_{n_1}^1\ ),\ ...,\ p_m(\ u_1^m,\ ...,\ u_{n_m}^m\ )$. La séquence d'instructions générées pour une telle question sera :

Allocate N

Sql( $p_1(\ u_1^1,\ ...,\ u_{n_1}^1\ )$ )

dont la dernière instruction est "call $p_1/n_1$

...

Sql( $p_m(\ u_1^m,\ ...,\ u_{n_m}^m\ )$ )

dont la dernière instruction est "call $p_m/n_m$ "

Deallocate

où N est le nombre de variables permanentes de la clause.

Voici les pseudo-codes modifiés :

**f. set_variable_temporary** $X_i$

Heap[H] $\longleftarrow$ ⟨ REF, h ⟩
$X_i$ $\longleftarrow$ Heap[H]
H $\longleftarrow$ H+1

**g. set_variable_permanent** $Y_i$

Heap[H] $\longleftarrow$ ⟨ REF, h ⟩
Stack[E+2+i] $\longleftarrow$ Heap[H]
H $\longleftarrow$ H+1

**h. set_value_temporary** $X_i$

Heap[H] $\longleftarrow$ $X_i$
H $\longleftarrow$ H + 1

**i. set_value_permanent** $Y_i$

Heap[H] $\longleftarrow$ Stack[E+2+i]
H $\longleftarrow$ H + 1

**j. put_variable_temporary** $X_n$, $A_i$

Heap[H] $\longleftarrow$ ⟨ REF, H ⟩
$X_n$ $\longleftarrow$ Heap[H]
$A_i$ $\longleftarrow$ Heap[H]
H $\longleftarrow$ H + 1

**k. put_variable_permanent** $Y_n$, $A_i$

Heap[H] ⟵ ⟨ REF, H ⟩
Stack[E+2+n] ⟵ Heap[H]
$A_i$ ⟵ Heap[H]
H ⟵ H + 1

**l. put_value_temporary** $X_n$, $A_i$

$A_i$ ⟵ $X_n$

**m. put_value_permanent** $Y_n$, $A_i$

$A_i$ ⟵ Stack[E+2+n]

**n. get_variable_temporary** $X_n$, $A_i$

$X_n$ ⟵ $A_i$

**o. get_variable_permanent** $Y_n$, $A_i$

Stack[E+2+n] ⟵ $A_i$

**p. get_value_temporary** $X_n$, $A_i$

Unify( $X_n$ , $A_i$ )

**q. get_value_permanent** $Y_n$, $A_i$

Unify( E+2+n , $A_i$ )

**r. unify_variable_temporary** $X_i$

Case mode of

| | |
|---|---|
| Read : | $X_i$ ⟵ Heap[S] |
| Write : | Heap[H] ⟵ ⟨ REF, H ⟩ |
| | $X_i$ ⟵ Heap[H] |
| | H ⟵ H + 1 |

Endcase
S ⟵ S + 1

**s. unify_variable_permanent** $Y_i$

Case mode of

| | |
|---|---|
| Read : | Stack[E+2+i] $\longleftarrow$ Heap[S] |
| Write : | Heap[H] $\longleftarrow$ $\langle$ REF, H $\rangle$ |
| | Stack[E+2+i] $\longleftarrow$ Heap[H] |
| | H $\longleftarrow$ H + 1 |

Endcase

S $\longleftarrow$ S + 1

**t. unify_value_temporary** $X_i$

Case mode of

| | |
|---|---|
| Read : | Unify( $X_i$, S ) |
| Write : | Heap[H] $\longleftarrow$ $X_i$ |
| | H $\longleftarrow$ H + 1 |

Endcase

S $\longleftarrow$ S + 1

**u. unify_value_permanent** $Y_i$

Case mode of

| | |
|---|---|
| Read : | Unify( E+2+i, S ) |
| Write : | Heap[H] $\longleftarrow$ Stack[E+2+i] |
| | H $\longleftarrow$ H + 1 |

Endcase

S $\longleftarrow$ S + 1

### 3.3.2 Exemples et implémentation des instructions

Exemples :

1. Soit le programme composé des procédures suivantes :
   p(X,Y):-q(X,Z),r(Z,Y).
   q(a,b).
   r(b,c).
   La question étant "?-p(U,V).", le code WAM généré sera :

```
p/2 : Allocate 2
      get_variable X3,A1
      get_variable Y1,A2
      put_value X3,A1
      put_variable Y2,A2
      call q/2
      put_value Y2,A1
      put_value Y1,A2
      call r/2
      Deallocate
q/2 : get_structure a/0 A1
      get_structure b/0 A2
      proceed
r/2 : get_structure b/0 A1
      get_structure c/0 A2
      proceed
      Allocate 0
      put_variable X3,A1
      put_variable X4,A2
      call p/2
      Deallocate
```

   Solution(s) ? oui, p(a,c).

2. Soit le programme composé des procédures suivantes :
   test :- p(X,X).
   p(X,f(X)).
   La question étant "?-test.", le code WAM généré sera :

```
test/0 : Allocate 0
         put_variable X3,A1
         put_value X3,A2
         call p/2
         Deallocate
p/2    : get_variable X3,A1
         get_structure f/1 A2
         unify_value X3
         proceed
         Allocate 0
         call test/0
         Deallocate
```

Solution(s) sans Occur-Check ? oui, test.
Solution(s) avec Occur-Check ? non, échec avec occur-check.

## Implémentation :

```
void W2_allocate ( unsigned n )
{
 unsigned long newE ;

 newE = E +Mem.Stack[E+2].mot +3 ;
 Mem.Stack[newE].mot = E ;
 Mem.Stack[newE+1].mot = CP ;
 Mem.Stack[newE+2].mot = n ;
 E = newE ;
 if (newE+2 > stack_max)
 { puts("! Memoire STACK depassee !");
   exit(1);
 }
}

void W2_deallocate()
{
 P = Mem.Stack[E+1].mot ;
```

```
 E = Mem.Stack[E].mot ;
}

void W2_set_variable_permanent(unsigned regy)
{
  Mem.Heap[H].mot=H*8+0;
  Mem.Stack[E+2+regy].mot=Mem.Heap[H].mot;
  H++;
}

void W2_set_variable_temporary(unsigned regx)
{
  Mem.Heap[H].mot=H*8+0;
  Mem.X[regx].mot=Mem.Heap[H].mot;
  H++;
}

void W2_set_value_permanent(unsigned regy)
{
  Mem.Heap[H].mot=Mem.Stack[E+2+regy].mot;
  H++;
}

void W2_set_value_temporary(unsigned regx)
{
  Mem.Heap[H].mot=Mem.X[regx].mot;
  H++;
}

void W2_unify_variable_permanent(unsigned regy)
{
  switch(mode)
  { case 'r' : Mem.Stack[E+2+regy].mot=Mem.Heap[S].mot;break;
    case 'w' : {
                 Mem.Heap[H].mot=(H*8)+0;
                 Mem.Stack[E+2+regy].mot=Mem.Heap[H].mot;
                 H++;
                 break;
```

```
                }
  }
  S++;
}

void W2_unify_variable_temporary(unsigned regx)
{
  switch(mode)
  { case 'r' : Mem.X[regx].mot=Mem.Heap[S].mot;break;
    case 'w' : {
                 Mem.Heap[H].mot=(H*8)+0;
                 Mem.X[regx].mot=Mem.Heap[H].mot;
                 H++;
                 break;
               }
  }
  S++;
}

char W2_unify_value_permanent(unsigned regy)
{
 char echec ;

 echec=0;
 switch(mode)
  { case 'r' : {
                 echec = W2_Unify(E+2+regy,S);
                 break;
               }
    case 'w' : {
                 Mem.Heap[H].mot=Mem.Stack[E+2+regy].mot;
                 H++;
                 break;
       }
  }
  S++;
  return(echec);
}
```

```
char W2_unify_value_temporary(unsigned regx)
{
 char echec ;

 echec=0;
 switch(mode)
  { case 'r' : {
                echec = W2_Unify(regx,S);
                break;
               }
    case 'w' : {
                Mem.Heap[H].mot=Mem.X[regx].mot;
                H++;
                break;
       }
  }
  S++;
  return(echec);
}

void W2_put_variable_permanent(unsigned regy,unsigned rega)
{
  unsigned long adr ;

  adr = E +regy +2 ;
  Mem.Stack[adr].mot=(adr)*8+0;
  Mem.X[rega].mot=Mem.Stack[adr].mot;
}

void W2_put_variable_temporary(unsigned regx,unsigned rega)
{
  Mem.Heap[H].mot=H*8+0;
  Mem.X[regx].mot=Mem.Heap[H].mot;
  Mem.X[rega].mot=Mem.Heap[H].mot;
  H++;
}
```

```
void W2_put_value_permanent(unsigned regy,unsigned rega)
{
  Mem.X[rega].mot=Mem.Stack[E+regy+2].mot;
}

void W2_put_value_temporary(unsigned regx,unsigned rega)
{
  Mem.X[rega].mot=Mem.X[regx].mot;
}

void W2_get_variable_permanent(unsigned regy,unsigned rega)
{
  Mem.Stack[E+2+regy].mot=Mem.X[rega].mot;
}

void W2_get_variable_temporary(unsigned regx,unsigned rega)
{
  Mem.X[regx].mot=Mem.X[rega].mot;
}

char W2_get_value_permanent(unsigned regy,unsigned rega)
{
  return(W2_Unify(E+2+regy,rega));
}

char W2_get_value_temporary(unsigned regx,unsigned rega)
{
  return(W2_Unify(regx,rega));
}

char W2_call(unsigned nom)
{
 if (nom == 0)
   return(1);
 else
 { CP = P + Mem.Code[P].cel.art ;
   P = Mem.Code[P+1].cel.nom ;
   return(0);
```

```
 }
}

void W2_proceed()
{
 P = CP ;
}
```

## 3.4 W.A.M. $\langle L_3, M_3 \rangle$

Le langage $L_3$ correspond au Prolog pur, étape finale de notre construction. Il s'agit d'une extension du langage $L_2$ permettant des définitions disjonctives, ce qui constitue le second niveau d'indexation des clauses. Plusieurs clauses peuvent ainsi partager le même nom de prédicat de tête. Dans ce langage, l'échec d'une unification ne provoque plus un arrêt net de l'exécution mais une prise en considération des alternatives de choix sur les clauses dans leur ordre d'apparition. Ceci se fait grâce au backtracking : le dernier point de choix est réexaminé en premier ( stratégie LIFO ).

Afin de permettre un bon déroulement du backtracking, il est nécessaire de sauver l'état de l'exécution au point de choix. Cette sauvegarde s'effectuera également sur le Stack, principalement dans un souci de protection des environnements alloués antérieurement au point de choix. Un nouveau registre global B indiquera une clause alternative pour un même prédicat.

Quand une clause est envisagée lors d'un point de choix, elle entraîne des liaisons de variables du Stack et du Heap. Celles-ci doivent être défaites lors de la reconsidération du choix. Nous allons conserver les adresses des variables qui doivent être déliées lors du backtracking dans une nouvelle et dernière table Trail. Cette table sera mise à jour lors de l'opération "Bind". Pour gérer le table, un registre global TR en indiquera la première cellule libre.

Lors du backtracking, les informations suivantes doivent être sauvegardées :

- les registres arguments $A_1$, ..., $A_n$ où n est l'arité du prédicat présentant des alternatives de choix ;
- le pointeur E de l'environnement courant du Stack ;
- le point CP de continuation de l'exécution ;
- le dernier point de choix B indiquant dans quelle direction doit se faire le backtracking en cas d'échec de toutes les alternatives ;
- l'étiquette vers la prochaine clause envisageable au point de choix ;
- le pointeur courant du Trail TR nécessaire pour défaire les liaisons lors du backtracking ;

- le sommet courant du Heap H nécessaire pour récupérer l'espace désalloué du Heap après un échec d'unification.

La structure d'un environnement de choix à conserver sera donc :

| | | |
|---|---|---|
| B : | n | ( nombre d'arguments ) |
| B+1 : | $A_1$ | ( $1^e$ registre argument ) |
| ... | ... | |
| B+n : | $A_n$ | ( $n^e$ registre argument ) |
| B+n+1 : | CE | ( point de l'environnement précédent ) |
| B+n+2 : | CP | ( point de continuation de l'exécution ) |
| B+n+3 : | B | ( point de choix précédent ) |
| B+n+4 : | BP | ( point de la prochaine clause ) |
| B+n+5 : | TR | ( pointeur du Trail ) |
| B+n+6 : | H | ( pointeur du Heap ) |

Afin de gérer au mieux le point de choix pour la multiple définition de clauses, trois nouvelles instructions sont introduites correspondant à la première — "try_me_else L" —, à l'intermédiaire — "retry_me_else L" — et à la dernière — "trust_me" — clause d'une définition, L étant l'adresse correspondante dans le Code.
Les effets de ces instructions sont respectivement :

- d'allouer un environnement de choix dans le Stack et d'affecter à L l'adresse de la prochaine instruction à envisager ;
- suite à un backtracking au point de choix courant ( indiqué par B ), de restaurer toutes les informations nécessaires et de mettre à jour L à l'adresse de la prochaine clause ;
- suite à un backtracking, de restaurer les informations nécessaires ainsi que le registre B à son prédecesseur.

Quant au backtracking, il doit permettre de restaurer le point de continuation P et de poursuivre l'exécution en cas de succès mais aussi, si aucun point de choix n'existe sur le Stack, d'échouer et d'interrompre l'exécution.
Deux autres opérations subordonnées "trail" et "unwind_trail" permettent de sauver et de restaurer toutes les variables depuis le dernier point de choix à un état 'désinstancié' ( libre ).

Une dernière remarque concernera une modification à apporter à l'instruction "Allocate N" de $L_2$. En effet, comme l'environnement de choix est ajouté au Stack, la gestion du

sommet du Stack se fait différemment suivant que cet environnement est le dernier ou non sur le Stack. Une comparaison sera ainsi introduite entre le registre E ( environnement d'allocation ) et le registre B ( environnement de choix ).

### 3.4.1 Spécification et pseudo-code

**a. Spécification des objets utilisés**

**B** : pointeur vers le début de l'environnement de choix courant.

**L** : adresse de la prochaine clause à envisager lors d'un point de choix.

**HB** : adresse permettant de récupérer l'espace désalloué dans le Heap.

**Trail** : pile des adresses des variables à "désinstancier" lors d'un backtracking.

**TR** : pointeur vers la première cellule libre du Trail.

**n** : arité du littéral courant

**num_of_args** : arité de la procédure courante au point de choix.

**b. Allocate N**

- O.U. : N, CP, E, B, Stack.
- Pré :

  $N \geq 0$
  $E_0$ pointe vers le début de l'environnement courant
  B pointe vers le début de l'environnement de choix s'il en existe un, sinon vers la base du Stack
  CP est le point de continuation de l'exécution
  $Stack_0$ est l'ensemble des environnements d'allocation et de choix

- Post :

  E pointe vers le nouvel environnement courant :
  E = $E_0$ + Stack[ $E_0$+2 ] + 3 si E > B
  E = B + Stack[ B ] + 7 si E $\leq$ B
  Stack = $\text{Stack}_0$ $\cup$ le nouvel environnement tel que :
  Stack[ E ] = $E_0$
  Stack[ E+1 ] = CP
  Stack[ E+2 ] = N
  Les N cellules suivantes sont réservées pour les variables permanentes

- Pseudo-code :

  if E > B then newE $\longleftarrow$ E + Stack[ E+2 ] + 3
  else newE $\longleftarrow$ B + Stack[ B ] + 7
  Stack[ E ] $\longleftarrow$ E
  Stack[ E ] $\longleftarrow$ CP
  Stack[ E ] $\longleftarrow$ N
  E $\longleftarrow$ newE

**c. try_me_else L**

- O.U. : L, E, Stack, B, CP, TR, H, HB, n, num_of_args, $A_i$.

- Pré :

  On choisit les instructions de la première clause d'une même définition.
  pour tout i, les registres $A_i$ arguments sont initialisés
  L est l'adresse de la première instruction de la prochaine clause dans le CODE
  E pointe vers le début de l'environnement courant
  $B_0$ pointe vers le début de l'environnement de choix s'il en existe un, sinon vers la base du Stack
  CP est le point de continuation de l'exécution
  $\text{Stack}_0$ est l'ensemble des environnements d'allocation et de choix
  H pointe vers la première cellule libre du Heap
  $\text{HB}_0$ pointe vers le début de l'espace à récupérer dans le Heap
  $\text{TR}_0$ pointe vers la première cellule libre du Trail

$n_0$ est l'arité du littéral précédent
num_of_args est l'arité de la procédure courante

- Post :

B = E + Stack[ E+2 ] + 3 si E > $B_0$
B = $B_0$ + Stack[ $B_0$ ] + 7 si E $\leq$ $B_0$
n = num_of_args
HB = H
Stack = $Stack_0$ $\cup$ le nouvel environnement de choix
$\forall$ $i$ Stack[ B+$i$ ] = $A_i$
Stack[ B+n+1 ] = E
Stack[ B+n+2 ] = CP
Stack[ B+n+3 ] = B
Stack[ B+n+4 ] = L
Stack[ B+n+5 ] = TR
Stack[ B+n+6 ] = H

- Pseudo-code :

if E > B then newB $\longleftarrow$ E + Stack[ E+2 ] + 3
else newB $\longleftarrow$ B + Stack[ B ] + 7
Stack[ newB ] $\longleftarrow$ num_of_args
n $\longleftarrow$ Stack[ newB ]
for $i$ $\longleftarrow$ 1 to n do Stack[ newB+$i$ ] $\longleftarrow$ $A_i$
Stack[ newB+n+1 ] $\longleftarrow$ E
Stack[ newB+n+2 ] $\longleftarrow$ CP
Stack[ newB+n+3 ] $\longleftarrow$ B
Stack[ newB+n+4 ] $\longleftarrow$ L
Stack[ newB+n+5 ] $\longleftarrow$ TR
Stack[ newB+n+6 ] $\longleftarrow$ H
B $\longleftarrow$ newB
HB $\longleftarrow$ H

**d. retry_me_else L**

- O.U. : L, E, Stack, B, CP, TR, H, HB, n, $A_i$.

- Pré :

  On choisit à présent les instructions de la clause suivante car un backtracking a eu lieu.
  L est l'adresse de la première instruction de la prochaine clause dans le CODE
  $E_0$ pointe vers le début de l'environnement courant
  $B_0$ pointe vers le début de l'environnement de choix s'il en existe un, sinon vers la base du Stack
  $CP_0$ est le point de continuation de l'exécution
  Stack est l'ensemble des environnements d'allocation et de choix
  $H_0$ pointe vers la première cellule libre du Heap
  $HB_0$ pointe vers le début de l'espace à récupérer dans le Heap
  $TR_0$ pointe vers la première cellule libre du Trail
  $n_0$ est l'arité du littéral précédent

- Post :

  n = Stack[ B ]
  $\forall\, i,\ 1 \leq i \leq$ n, $A_i$ = Stack[ B+$i$ ]
  $\forall\, i$, Stack[ B+n+5 ] $\geq i \geq$ TR, Trail[ $i$ ] = $\langle$ REf, Trail[ $i$ ] $\rangle$
  E = Stack[ B+n+1 ]
  CP = Stack[ B+n+2 ]
  TR = Stack[ B+n+5 ]
  H = Stack[ B+n+6 ]
  HB = H
  Stack inchangé et B inchangé

- Pseudo-code :

  n $\longleftarrow$ Stack[ B ]
  for $i$ $\longleftarrow$ 1 to n do $A_i$ $\longleftarrow$ Stack[ newB+$i$ ]
  E $\longleftarrow$ Stack[ B+n+1 ]
  CP $\longleftarrow$ Stack[ B+n+2 ]
  Stack[ newB+n+4 ] $\longleftarrow$ L
  unwind_trail( B+n+5, TR )
  Stack[ newB+n+5 ] $\longleftarrow$ TR
  Stack[ newB+n+6 ] $\longleftarrow$ H
  HB $\longleftarrow$ H

**e. trust_me**

- O.U. : E, Stack, B, CP, TR, H, num_of_args.
- Pré :

  L est l'adresse de la première instruction de la prochaine clause dans le CODE
  $E_0$ pointe vers le début de l'environnement courant
  $B_0$ pointe vers le début de l'environnement de choix s'il en existe un, sinon vers la base du Stack
  $CP_0$ est le point de continuation de l'exécution
  Stack est l'ensemble des environnements d'allocation et de choix
  $H_0$ pointe vers la première cellule libre du Heap
  $HB_0$ pointe vers le début de l'espace à récupérer dans le Heap
  $TR_0$ pointe vers la première cellule libre du Trail
  $n_0$ est l'arité du littéral précédent

- Post :

  n = Stack[ $B_0$ ]
  $\forall\, i,\ 1 \leq i \leq$ n, $A_i$ = Stack[ B+$i$ ]
  $\forall\, i$, Stack[ B+n+5 ] $\geq i \geq$ TR, Trail[ $i$ ] = $\langle$ REf, Trail[ $i$ ] $\rangle$
  n = Stack[ $B_0$ ]
  E = Stack[ $B_0$+n+1 ]
  CP = Stack[ $B_0$+n+2 ]
  TR = Stack[ $B_0$+n+5 ]
  H = Stack[ $B_0$+n+6 ]
  B = Stack[ $B_0$+n+3 ]
  HB = Stack[ B+N+6 ]
  Stack inchangé

- Pseudo-code :

  n $\longleftarrow$ Stack[ B ]
  for $i \longleftarrow$ 1 to n do $A_i \longleftarrow$ Stack[ newB+$i$ ]
  E $\longleftarrow$ Stack[ B+n+1 ]
  CP $\longleftarrow$ Stack[ B+n+2 ]

Stack[ newB+n+4 ] ⟵ L
unwind_trail( B+n+5, TR )
TR ⟵ Stack[ B+n+5 ]
H ⟵ Stack[ B+n+6 ]
B ⟵ Stack[ B+n+3 ]
HB ⟵ Stack[ newB+n+6 ]

**f. backtrack**

- O.U. : P, B, Stack.
- Pré :

  $P_0$ pointe vers la dernière instruction d'une clause, en fin de branche de l'arbre de recherche soit avec succès soit avec échec..

- Post :

  P pointe vers la prochaine instruction à exécuter de la branche la plus proche au dernier point de choix.
  P = Stack[ B + Stack[ B ] + 4 ]

- Pseudo-code :

  P ⟵ Stack[ B+Stack[ B ]+4]

**h. trail( a : adresse )**

- O.U. : P, B, Stack.
- Pré :

  a repr $t_a$
  $Trail_0$ est la table des variables à "désinstancier"
  $TR_0$ pointe vers la première cellule libre du Trail

- Post :

Si la construction de $t_a$ est antérieure au point de choix, alors il faut sauver $t_a$ dans le Trail
Trail = $\text{Trail}_0 \cup t_a$
$\text{TR} = \text{TR}_0 + 1$

- Pseudo-code :

if ( a < HB ) $\vee$ (( H < a ) $\wedge$ ( a < B )) then
    Trail[ TR ] $\longleftarrow$ a
    TR $\longleftarrow$ TR + 1

**i. unwind_trail ( $a_1$, $a_2$ : adresses )**

- O.U. : P, B, Stack.
- Pré :

$\forall i,\ a_1 \leq i \leq a_2,\ Trail[i]$ repr $t_i$
les variables $t_i$ doivent être restaurées dans les zones Heap, Stack et X (registres).

- Post :

$\forall i,\ a_1 \leq i \leq a_2$, Store[ Trail[ i ] ] = $\langle$ REF, Trail[ $i$ ] $\rangle$

- Pseudo-code :

for i $\longleftarrow$ $a_1$ to $a_2$ do Store[ Trail[ i ] ] $\longleftarrow$ $\langle$ REF, Trail[ i ] $\rangle$

**j. Bind( $d_1$, $d_2$ : adresses)**

- Les spécifications antérieures restent inchangées. Seul le Trail viendra s'ajouter aux objets déjà utilisés.
- Pseudo-code :

$\langle tag_1, - \rangle \longleftarrow$ Store [ $d_1$ ]
$\langle tag_2, - \rangle \longleftarrow$ Store [ $d_2$ ]
if ($tag_1$ = REf) and (($tag_2 \neq$ REF) or ($d_2 < d_1$))
then Store [ $d_1$ ] $\longleftarrow$ Store [ $d_2$ ] ; trail( $d_1$ )
else Store [ $d_2$ ] $\longleftarrow$ Store [ $d_1$ ] ; trail( $d_2$ )

### k. Code des instructions

| Instruction | Code opératoire | nombre d'opérandes | taille de l'instruction |
|---|---|---|---|
| "fin" | 0 | 0 | 1 |
| proceed | 1 | 0 | 1 |
| call | 2 | 1 | 2 |
| put_structure | 3 | 3 | 3 |
| put_variable_temporary | 4 | 2 | 2 |
| put_variable_permanent | 5 | 2 | 2 |
| put_value_temporary | 6 | 2 | 2 |
| put_value_permanent | 7 | 2 | 2 |
| get_structure | 8 | 3 | 3 |
| get_variable_temporary | 9 | 2 | 2 |
| get_variable_permanent | 10 | 2 | 2 |
| get_value_temporary_occur | 11 | 2 | 2 |
| get_value_permanent_occur | 12 | 2 | 2 |
| set_variable_temporary | 13 | 1 | 2 |
| set_variable_permanent | 14 | 1 | 2 |
| set_value_temporary | 15 | 1 | 2 |
| set_value_permanent | 16 | 1 | 2 |
| unify_variable_temporary | 17 | 1 | 2 |
| unify_variable_permanent | 18 | 1 | 2 |
| unify_value_temporary_occur | 19 | 1 | 2 |
| unify_value_permanent_occur | 20 | 1 | 2 |
| allocate | 21 | 1 | 2 |
| deallocate | 22 | 0 | 1 |
| try_me_else | 23 | 1 | 2 |
| retry_me_else | 24 | 1 | 2 |
| trust_me | 25 | 0 | 1 |

### 3.4.2 Exemples et implémentation des instructions

Exemples :

1. Soit le programme composé des procédures suivantes :
   a:-b(X),c(X).
   b(X):-e(X).
   c(1).
   e(X):-f(X).
   e(X):-g(X).
   f(2).
   g(1).
   La question étant "?-a.", le code WAM généré sera :

```
      a/0 : Allocate 1
            put_variable Y1,A1
            call b/1
            put_value Y1,A1
            call c/1
            Deallocate
      b/1 : Allocate 0
            get_variable X2,A1
            put_value X2,A1
            call e/1
            Deallocate
      c/1 : get_structure 1/0 A1
            proceed
   L1:e/1 : Allocate 0
            get_variable X2,A1
            put_value X2,A1
            call f/1
            Deallocate
   L2:e/1 : Allocate 0
            get_variable X2,A1
            put_value X2,A1
```

```
        call g/1
        Deallocate
f/2 :   get_structure 2/0 A1
        proceed
g/1 :   get_structure 1/0 A1
        proceed
        Allocate 0
        call a/0
        Deallocate
```

Solution(s) ? oui, a.

2. Soit le programme bien connu composé des procédures suivantes :
   append(nil,X,X).
   append(cons(T,X),Y,cons(T,Z):-append(X,Y,Z).
   Le code WAM pour ce programme sera :

```
L1:append/3 :   get_structure nil/0 A1
                get_variable X4,A2
                get_value X4,A3
                proceed
L2:append/3 :   Allocate 0
                get_structure cons/2 A1
                unify_variable X4
                unify_variable X5
                get_variable X6,A2
                get_structure cons/2 A3
                unify_value X4
                unify_variable X7
                put_value X5,A1
                put_value X6,A2
                put_value X7,A3
                call append/3
                Deallocate
```

   Si la question est : "?-append(X,Y,cons(a,cons(b,nil))).”
   Le code WAM correspondant sera :

```
Allocate 0
put_variable X4,A1
put_variable X5,A2
put_structure a/0 X6
put_structre b/0 X8
put_strucre nil/0 X9
put_structure cons/2 X7
set_value X8
set_value X9
put_structure cons/2 A3
set_value X6
set_value X7
call append/3
Deallocate
```

Solution(s) ? oui, append(nil,cons(a,cons(b,nil)),cons(a,cons(b,nil))).
oui, append(cons(a,nil),cons(b,nil),cons(a,cons(b,nil))).
oui, append(cons(a,cons(b,nil)),nil,cons(a,cons(b,nil))).

Si la question est : "?-append(X,Y,Z)."
Le code WAM correspondant sera :

```
Allocate 0
put_variable X4,A1
put_variable X5,A2
put_variable X6,A3
call append/3
Deallocate
```

Solution(s) ?
oui, append(nil,Var_22,Var_22).
oui, append(cons(Var_26,nil),Var_22,cons(Var_26,Var_22)).
oui, append(cons(Var_26,cons(Var_34,nil)), Var_22,cons(Var_26,cons(Var_34,Var_22))).
oui, ...
! Stop : Mémoire WAM dépassée !

## Implémentation :

```
void W3_try_me_else(unsigned L)
{
  unsigned i;
  unsigned long newB ;

  if (E > B)
    newB=E+Mem.Stack[E+2].mot+3;
  else
    newB=B+Mem.Stack[B].mot+7;

  Mem.Stack[newB].mot=Num_of_args;
  N=Mem.Stack[newB].mot;
  for(i=1;i<=N;i++)
    Mem.Stack[newB+i].mot=Mem.X[i].mot;
  Mem.Stack[newB+N+1].mot=E;
  Mem.Stack[newB+N+2].mot=CP;
  Mem.Stack[newB+N+3].mot=B;
  Mem.Stack[newB+N+4].mot=L;
  Mem.Stack[newB+N+5].mot=TR;
  Mem.Stack[newB+N+6].mot=H;
  B=newB;
  HB=H;
}

void W3_retry_me_else(unsigned L)
{ unsigned i ;

  N=Mem.Stack[B].mot;
  for(i=1;i<=N;i++)
    Mem.X[i].mot = Mem.Stack[B+i].mot ;
  E = Mem.Stack[B+N+1].mot ;
  CP = Mem.Stack[B+N+2].mot ;
  Mem.Stack[B+N+4].mot = L ;
  W3_unwind_trail(Mem.Stack[B+N+5].mot,TR) ;
  TR = Mem.Stack[B+N+5].mot ;
```

```
  H = Mem.Stack[B+N+6].mot ;
  HB = H ;

}

void W3_trust_me ()
{
 unsigned i ;

 N = Mem.Stack[B].mot ;

 for(i=1;i<=N;i++)
   Mem.X[i].mot = Mem.Stack[B+i].mot ;
 E = Mem.Stack[B+N+1].mot ;
 CP = Mem.Stack[B+N+2].mot ;
 W3_unwind_trail(Mem.Stack[B+N+5].mot,TR) ;
 TR = Mem.Stack[B+N+5].mot ;
 H = Mem.Stack[B+N+6].mot ;
 B = Mem.Stack[B+N+3].mot ;
 HB = Mem.Stack[B+N+6].mot ;
}

void W3_unwind_trail(  long a1,  long a2)
{
 unsigned i ;

 for(i=a1;i<=(a2-1);i++)
   Mem.Store[Mem.Trail[i].mot] = Mem.Trail[i].mot*8 + 0 ;
}

void W3_trail(unsigned long a)
{
 if ( (a<HB) || ((H<a)&&(a<B)) )
  {
   Mem.Trail[TR].mot=a ;
   TR++ ;
  }
}
```

```
char backtrack()
{
 if (B == heap_max)
  return(1) ;
 else
  {
   P = Mem.Stack[B+Mem.Stack[B].mot+4].mot ;
   return(0) ;
  }
}

void W3_allocate ( unsigned n )
{
 unsigned long newE ;

 if(E>B)
   newE = E +Mem.Stack[E+2].mot +3 ;
 else
   newE = B + Mem.Stack[B].mot + 7 ;

 Mem.Stack[newE].mot = E ;
 Mem.Stack[newE+1].mot = CP ;
 Mem.Stack[newE+2].mot = n ;
 E = newE ;
 if (newE+2 >= stack_max)
 { puts("! Memoire STACK depassee !");
   exit(1);
 }
}

char W3_call(unsigned nom)
{
 char echec ;

 if (nom == 0)
 {
  echec = backtrack() ;
```

```
  return( echec ) ;
 }
 else
 { CP = P + Mem.Code[P].cel.art ;
   Num_of_args =  tabfonct[Mem.Code[P+1].cel.art].arite ;
   P = Mem.Code[P+1].cel.nom ;
   return(0);
 }
}


void W3_Bind(unsigned long a1,unsigned long a2)
{
 char echec,t1,t2;

 t1 = type ( Mem.Store[a1] );
 t2 = type ( Mem.Store[a2] );
 if ((t1==0)&&((t2!=0)||(a2<a1)))
 { Mem.Store[a1]=Mem.Store[a2];
   W3_trail(a1) ;
 }
 else
 { Mem.Store[a2]=Mem.Store[a1];
   W3_trail(a2) ;
 }
}
```

# Chapitre 4

# Prolongements envisageables

## 4.1 Optimisation de la Machine de Warren

Dans le chapitre précédent, nous avons présenté la machine pure de Warren. Nombreuses sont les optimisations qu'il reste à apporter pour obtenir la machine complète. Nous ne les réaliserons pas dans ce travail. Cependant, dans un souci de perspective ultérieure, nous allons brosser rapidement quelques unes des optimisations intéressantes à envisager.

Trois principes régissent l'optimisation de la machine pure :

1. Il importe de veiller à une utilisation parcimonieuse du Heap notamment au niveau de la construction des termes relativement persistants.
2. L'allocation des registres doit se faire de façon à éviter tout mouvement ultérieur inutile et à minimiser la taille du code.
3. Certaines situations particulières se rencontrant souvent, il peut être intéressant, même si elles sont déjà prises en charge par des instructions générales, de les assumer par des instructions plus spécifiques permettant un gain de temps ou d'espace.

### 4.1.1 Représentation du Heap

Certains auront sans doute remarqué l'importante indirection dans le Heap, notamment sur les structures. En fait, il n'est pas nécessaire d'allouer systématiquement une cellule STR précédant la cellule de chaque foncteur.
Il en résulte un gain de place mais aussi l'élimination d'un niveau d'indirection et l'accroissement simultané de la vitesse de déréférenciation. En particulier sur l'exemple p(Z,h(Z,W),f(W)), nous pouvons observer un gain de trois cellules.

| | | |
|---|---|---|
| 0 : | h | 2 |
| 1 : | REF | 1 |
| 2 : | REF | 2 |
| 3 : | f | 1 |
| 4 : | REF | 2 |
| 5 : | p | 3 |
| 6 : | REF | 1 |
| 7 : | STR | 0 |
| 8 : | STR | 3 |

Remarquons que si cette modification est adoptée, il faudrait revoir les représentations mises en place au chapitre 2 car la seule adresse pouvant représenter p(Z,h(Z,W),f(W)) est 5, or la cellule d'adresse 5 n'est pas du type

| | | |
|---|---|---|
| $\alpha$ : | STR | $\beta$+1 |

D'où la remise en cause des représentations introduites alors.

### 4.1.2 Constantes, listes et variables anonymes

En Prolog, des structures particulières telles que listes, constantes, ... se rencontrent bien souvent. Même si notre machine pure admet déjà de telles structures, les instructions pour les manipuler sont assez lourdes. En accord avec le troisième principe, des instructions spéciales se chargeront des structures 0-aires ( les constantes ), des listes et des variables dites anonymes. Ces dernières sont des variables dont la portée est limitée à un littéral.
Pour les listes et les constantes, on optimise les instructions "get_", "put_" et "unify_". D'une part, on leur assigne des nouveaux types, respectivement, LIS et CON, qui s'ajouteront aux deux précédents REF et STR. D'autre part, apparaissent des instructions telles que "unify_constant a", "put_list $X_i$" et "get_list $X_i$".
Quant aux variables dites anonymes, ces variables n'ayant qu'une seule occurrence, il n'est pas nécessaire de leur associer des registres. Les instructions "set_void", "unify_void" leur seront associées.

Ainsi, une séquence d'instruction du type :

Unify_variable $X_i$
Set_structure a/0 $X_i$

peut être remplacé par :

Unify_constant a/0 $X_i$

Parallèlement à la spécialisation des instructions, un nouveau type peut être placé dans le tag des cellules : CON.
La cellule

$\alpha$ : | CON | a |

remplacera dès lors

$\alpha$ : | STR | $\beta$+1 | et $\beta$+1 : | a | 0 |

Pour les listes, selon [AK91], les foncteurs de listes non vides n'ont pas besoin d'être construits sur le Heap. Les listes seront construites de sorte que le type LIS indique que la partie adresse correspondante détermine la cellule du premier élément d'une paire. Par convention de contigüité sur les sous-termes, le second de la paire indique toujours l'adresse du suivant dans la liste. L'atome [] sert de symbole de fin de listes. Ainsi f([Z,W]) sera construit comme suit :

| | | | | |
|---|---|---|---|---|
| Heap | 0: | REF | 0 | 0 repr W |
| | 1: | CON | [] | 1 repr [] |
| | 2: | REF | 2 | 2 repr Z |
| | 3: | LIS | 0 | 3 repr [W\|[]] |
| A | 1: | LIS | 2 | $A_1$ repr [Z,W] i.e. [Z\|[W\|[]]] |

Soit le pseudo-code de put_list $X_i \equiv X_i \longleftarrow \langle$ LIS, H $\rangle$, le code généré pour ce terme sera donc :

```
put_list X3
set_variable X4
set_const []
put_list A1
set_variable X2
set_value X3
```

Dans p(Z,h(Z),f(W)), W est une variable dite anonyme. Il n'est pas utile de lui associer un registre. Il suffit de construire une variable non liée sur le Heap. Quant aux variables anonymes qui sont des arguments dans la tête d'une clause, elles peuvent être simplement ignorées. Ainsi l'instruction "get_variable $X_3$, $A_1$" est inutile pour un fait tel que p(_,g(X),f(_,Y,_)) où "_" signale la présence d'une variable anonyme déclarée explicitement. Le code généré est :

```
p/3 : get_structure g/1 A2
```

```
unify_void 1
get_structure f/3 A3
unify_void 3
```

### 4.1.3 Allocation de registres

Même grâce aux optimisations sur les variables anonymes, il reste des instructions telles que "get_variable $X_3$, $A_1$" qui s'avèrent inutiles. En effet, par l'optimisation précédente, les seules variables temporaires qu'il nous reste sont celles apparaissant dans la tête et le premier but, toutes les autres variables temporaires étant anonymes. Pour l'instruction "get_variable $X_3$, $A_1$" le registre $X_3$ n'étant qu'une copie de $A_1$ n'est pas indispensable. Nous pourrions gagner un registre si nous améliorons l'instruction par "get_variable $A_1$, $A_1$" ce qui revient à ne rien faire ( en effet $A_1 \longleftarrow A_1$). Nous pouvons donc éliminer complètement ce type d'instructions. Il en ressort un gain non seulement d'espace mais aussi de temps grâce aux mouvements des registres évités.

### 4.1.4 Les clauses à un but

Nous retrouvons ici la remarque établie dans $\langle L_2, M_2 \rangle$ au sujet de l'instruction "Allocate N" quand N = 0. En effet, comme il est certain que toutes les variables d'une clause à un but sont temporaires, les instructions "Allocate 0" et "Deallocate" peuvent être éliminées. Donc, pour une clause $p(X_1, ..., X_n) : -q(Y_1, ..., Y_m)$ dont le code était

p: Allocate 0
compilation de l'atome $p$
compilation de l'atome $q$
Call $p$/n
Deallocate

il devient à présent

p: compilation de l'atome $p$
compilation de l'atome $q$
Call $p$/n

On peut généraliser cela à toutes les clauses sans variables permanentes. Les conséquences en sont un gain de place dans le Stack et un gain de temps car nous évitons une sauvegarde et une restauration inutiles.

## 4.2 Interprétation abstraite

Dans cette partie, nous allons aborder l'analyse statique des programmes Prolog par l'interprétation abstraite. Dans le cadre de ce travail, nous exposerons brièvement en quoi consiste cette méthode. Nous exposerons comment cette technique peut s'appliquer aux programmes Prolog. Notre ambition n'étant ici que de faire un bref tour d'horizon, nous n'exposerons que les principes. Les algorithmes pour mettre en oeuvre ces principes ainsi que la théorie mathématique sous-jacente ne seront pas abordés. Des ouvrages spécialisés traitent de ce sujet. Parmi ceux-ci, citons notamment le rapport [LC91].

En quoi consiste une telle analyse? Le principe de base de la méthode est d'exécuter le programme sur un domaine abstrait plutôt que de l'appliquer au domaine de calcul habituel ( ou standard ). Ce domaine abstrait doit cependant vérifier deux hypothèses. Premièrement, les éléments du domaine abstrait doivent représenter des propriétés utiles du domaine standard. La seconde hypothèse est la convergence en un temps fini et l'efficacité des calculs sur le domaine abstrait. Par exemple, considérons un programme qui doit effectuer une multiplication entre deux nombres réels. Le domaine de calcul habituel est l'ensemble des nombres réels. Nous pouvons remplacer ce domaine par le domaine abstrait suivant : $\{+, -\}$. Ainsi, la multiplication de deux nombres peut être représentée par les règles de signes : $+.+ = +$, $-.- = +$, $+.- = -$ et $-.+ = -$. Ces règles permettent de calculer le signe d'un produit sans effectuer de calcul.

Le but de cette analyse est de transformer un programme pour l'optimiser. Nous pouvons citer par exemple la transformation de la récursivité en boucle ou encore le déplacement à l'exterieure d'une boucle d'une expression dont la valeur reste constante lors de l'exécution du corps de la boucle. Cette méthode permet aussi de vérifier la correction des programmes par rapport à certains critères comme par exemple la vérification du caractère bien typé des expressions dans le programme.

Beaucoup d'applications de l'analyse statique sont possibles et cela quelque soit le langage de programmation. Pour Prolog, nous pouvons constater qu'énormément de recherches sont actuellement effectuées dans ce sens, preuve s'il en est que le domaine est porteur de beaucoup d'espoirs. Une des tâches de l'interprétation abstraite pour les systèmes Prolog est la génération de programmes plus efficaces et plus compacts. Montrons le sur quelques exemples.

Le langage Prolog possède une caractéristique importante appelée la multidirectionnalité. Cela signifie que tout paramètre d'une procédure peut-être utilisé tantôt comme donnée, tantôt comme résultat. Par exemple, la même procédure *longueur* (LIST,L) pourra servir à donner la longueur d'une liste, à tester si une liste est d'une longueur donnée ou à engendrer l'ensemble des listes de longueur donnée. L'étendue des utilisations possibles des procédures Polog peut être vaste. L'inconvénient majeur est le manque d'efficacité de telle procédure. Cependant, en pratique, les programmes n'utilisent pas cette propriété de multidirectionnalité. Cela est une porte ouverte à la détection par l'analyse statique des utilisations qui sont faites des procédures et à la génération de codes plus efficaces.

Une telle analyse ne se réalise pas d'un seul coup. Elle nécessite des résultats d'autres analyses comme par exemple celle qui détecte les procédures déterministes. Celles-ci sont des procédures dont on est sûr qu'elles donneront toujours le même résultat. La détection de telles procédures permet de remplacer leur code par le résultat qu'elles sont susceptibles de donner. Le programme y gagne ainsi en espace et en efficacité.

Une autre application possible de l'interprétation abstraite pour les systèmes Prolog est la détection des tests d'occurrence ( occur-check ). Nous savons qu'il est interdit d'unifier une variable avec un terme la contenant. Cela signifie qu'à chaque unification, il est nécessaire d'effectuer un test d'occurrence. Or ce test coûte cher. De plus, nous constatons que dans la réalité, l'occur-check survient rarement dans les programmes. Un des buts que vise l'analyse statique du programme est la suppression de ce test.

D'autres applications sont bien sûr possibles. Mais ce bref tour d'horizon de ce qu'est l'interprétation abstraite et de son utilité aura montré au lecteur la nécessité d'une telle méthode et les espoirs dont elle est porteuse.

Concernant l'optimisation de la machine de Warren, nous présentons une application de l'interprétation abstraite sur la génération des instructions WAM et la simplification de l'algorithme d'unification. L'interprétation abstraite permet la détection de situations telles que $X_1 = \mathrm{f}(\ X_2, ..., X_n\ )$ ou $X_1 = X_2$, où $X_1$ est une variable. Dans de telles situations, nous constatons qu'il est possible de simplifier l'algorithme d'unification "Unify" en évitant la création d'une pile et les différents tests. L'algorithme se reduit alors à

$$\mathrm{Bind}(\ \mathrm{Deref}(\alpha_1),\ \alpha_2\ )$$

où $t_{\alpha_1}$ représente le membre de gauche et $t_{\alpha_2}$ celui de droite. La simplification porte sur les instructions WAM "get_value $X_n$, $A_i$" et "unify_value $X_i$" qui réalisent toutes les deux l'algorithme d'unification par un appel à "Unify". Notons que "unify_value $X_i$" n'opère

l'unification qu'en mode Read.
Nous introduisons le pseudo-code de deux nouvelles instructions

unify_bind $X_i \equiv$ Bind( Deref($X_i$), S))
get_bind $X_n$, $A_i \equiv$ Bind( Deref($X_n$), $A_i$))

Illustrons cette application.
Soit le fait k(p(Z,h(W)),p(Y,Z)) et la question ?-k(U,U).
Le code généré est le suivant :

```
k/2 : get_structure p/2 A1
      unify_variable X3
      unify_variable X4
      get_structure h/1 X4
      unify_variable X5
      get_structure p/2 A2
      unify_variable X6
      unify_value X3
      proceed

      put_variable X3 A1
      put_value X3 A2
      call k/2
```

Lors du "get_structure p/2 $A_2$", nous passons en mode Read [1], ce qui engendre que l'instruction "unify_value $X_3$" se traduit par un appel à "Unify". Nous pouvons remplacer cette instruction par la nouvelle instruction "unify_bind $X_3$". Cette dernière instruction réalise l'unification de la variable $t_{X_3}$ ( Z ) et du terme $t_S$ ( h(W) ) sans avoir recours à l'algorithme d'unification.

[1] en effet, le squelette du foncteur p/2 est déjà construit sur le Heap

# Chapitre 5

# L'analyseur syntaxique

Dans les chapitres précédents, nous avons vu comment compiler un programme Prolog. De manière à rendre cette compilation automatique, nous avons rédigé un analyseur pour chaque machine. Chaque analyseur est un programme qui réalise la tokenization des clauses du programme et les transforme en instructions WAM.

Dans ce chapitre, nous allons expliquer le contenu de ces différents analyseurs syntaxiques, étant entendu que l'objectif final de tels programmes est l'interprétation des langages $L_i$ des machines $(M_i)_{0 \leq i \leq 3}$, en instructions WAM. Le but de ce chapitre se veut documentaire pour le lecteur qui a l'intention de s'investir dans la compréhension de ces analyseurs. Nous supposerons que le lecteur connaît les caractéristiques des différentes machines. Nous supposerons aussi qu'il est capable d'engendrer les instructions qui permettent la compilation d'un terme, d'un fait ou d'une clause.

## 5.1 L'analyseur de WAM 0

Cette partie est consacrée à l'analyseur de la machine $\langle L_0, M_0 \rangle$. Nous y aborderons successivement les conventions que nous avons prises, les structures de données que nous utilisons, la lecture et la tokenization d'un terme sur un exemple et l'exécution des instructions WAM.

### 5.1.1 Conventions

Nous supposons lors de la compilation d'un terme que :

- le nom d'une variable ne dépasse pas lg_var_max(=25) caractères;
- le nom d'un foncteur ou d'une constante ne dépasse pas lg_fonct_max(=20) caractères;
- la longueur maximale de la chaîne représentant le terme à compiler ne dépasse pas lg_str_max(=80) caractères;
- le nombre de variables différentes qui apparaissent dans le programme et la question est au plus de nb_var(=50);
- le nombre de foncteurs ou de constantes ( y compris les répétitions ) ne dépasse pas nb_fonct(=100);
- l'arité maximale d'un foncteur ne dépasse pas arite_max(=20).

Il est évident que l'utilisateur peut changer à sa guise ces conventions en allant modifier les constantes définies dans le fichier source ANVAR.H.

### 5.1.2 Structures de données

La définition complète de ces structures est reprise dans le fichier ANVAR.H de WAM 0. Nous ne les reprendrons donc plus ici; nous nous contenterons seulement de les décrire.

**s** : Il nous faut lire une chaîne de caractères qui représente une question ou un programme. Cette chaîne sera toujours contenue dans la variable *s* qui sera vue comme un tableau de caractères.

**ch** : Pour pouvoir analyser cette chaîne, il nous faudra la décomposer en sous-chaînes qui seront, soit le nom d'un foncteur, d'une constante ou d'une variable, soit un

caractère spécial appartenant à l'ensemble des caractères suivants : ( ) , : ? - . et \0. la variable *ch* contiendra la sous-chaîne courante.

**lettre** : La variable *ch* ne sera qu'une copie d'une partie de *s*. Le rôle de *lettre* est de savoir à quelle position de *s* nous sommes arrivés dans le décodage du terme.

**index** : Cette variable contiendra en permanence l'indice du premier registre libre qui peut être assigné à une variable, à un foncteur ou à une constante. On pourrait trouver bizarre le fait que cette variable soit initialisée à deux. La raison en est simple. Le foncteur de tête du terme principal occupera toujours le premier registre. Celui-ci étant donc virtuellement pris, le registre suivant qui est libre est le deuxième.

**util** : Nous savons qu'il y a dans la machine $M_0$ trois instructions qui permettent de recopier la valeur des registres dans le Heap, à savoir Set_value, Get_value et Unify_value. Nous avons aussi trois instructions, Set_variable, Get_variable et Unify_variable, qui permettent d'initialiser ces registres. Ces trois dernières instructions sont utilisées lorsque nous assignons pour la première fois une variable à un registre. les trois premières sont utilisées à chaque nouvel usage de ces registres. Le but de ce tableau est de savoir si oui ou non, nous avons déjà rencontré ce registre lors de la tokenization. Un flag à un à la $i^{eme}$ position dans *util* indique que le $i^{eme}$ registre est assigné à une variable. A toute nouvelle occurence de cette variable, il suffira de recopier le contenu de ce registre. Ainsi par exemple si nous avons que *s* représente p(f(X),Y,X,g(Z)), alors util[1]=0, util[2]=0, util[3]=1, util[4]=1, util[5]=0, util[6]=0, util[7]=1.

**tabvar** : Comme son nom l'indique, ce tableau contient l'ensemble des noms des variables différentes rencontrées lors de la compilation du programme et de la question ainsi que le numéro du registre qui est attribué à chaque variable. D'où, lorsqu'une variable ( dont le nom est contenu dans *ch* ) est décodée, la consultation du tableau nous permet de savoir si elle a déjà été vue, auquel cas un registre lui a déjà été alloué. Dans le cas où elle n'aurait pas été rencontrée, la variable sera ajoutée au tableau et son numéro de registre sera la valeur de *index*.

**tabfonct** : Comme son nom l'indique aussi, ce tableau contiendra l'ensemble des noms de foncteurs et de constantes. On y fera référence par un pointeur ( qui dans le programme commence par itf ). Ce tableau a dans la machine $M_0$ une structure simple qui est appelée à évoluer par la suite.

**listeS** : C'est le tableau le plus important. Il va contenir l'ensemble des informations sur le terme à décomposer. Il permettra aussi sa compilation. Ce tableau possède trois champs : *type*, *reg*, *pos*. Le premier d'entre eux, *type*, permet de savoir quel type de

structure nous avons : 0 si c'est une variable, 1 si c'est un foncteur ou une constante. On pourrait dans des versions plus évoluées, envisager d'ajouter le type constante, le type liste ou tout autre type. Le champ *reg* contient le numéro du registre qui est assigné à la variable, au foncteur ou à la constante. Enfin le dernier champ, *pos*, est un pointeur. Il est nul si le type représente une variable. S'il s'agit d'un foncteur ou d'une constante, *pos* contient l'indice dans *s* de la première lettre du nom du foncteur ou de la constante. Comme pour le tableau *tabfonct*, cette structure de donnée est appelée à évoluer par la suite.

Nous avons mis l'ensemble de ces variables en global, exeption faite de *listeS*; ceci afin de ne pas avoir toujours à les passer comme paramètres.

### 5.1.3 Lecture et tokenization d'un terme

Dans cette partie, nous allons expliquer sur un exemple comment fonctionne notre algorithme. L'idée de départ est de simuler la tokenization des termes et de les compiler suivant le schéma ainsi obtenu. Soit donc le terme p(f(X),h(Y,f(a)),Y) à compiler. Sa tokenization est la suivante :

$$\begin{aligned} X_1 &= \mathrm{p}(X_1,X_2,X_3) \\ X_2 &= \mathrm{f}(X_5) \\ X_3 &= \mathrm{h}(X_4,X_6) \\ X_4 &= \mathrm{Y} \\ X_5 &= \mathrm{X} \\ X_6 &= \mathrm{f}(X_7) \\ X_7 &= \mathrm{a} \end{aligned}$$

Nous allons montrer que l'algorithme construit une représentation de cette tokenization. Nous montrerons aussi comment nous pouvons nous servir de cette représentation pour compiler le terme.

Initialement,voici l'état de nos structures de données :

**lettre** = 0
**index** = 2
**util**[$i$] = 0, $\forall i$
**tabfonct**[$i$] = 0, $\forall i$
s = p(f(X),h(Y,f(a)),Y).
les champs de **listeS**[$i$] = 0, $\forall i$
les champs de **tabvar**[$i$] = 0, $\forall i$

On commence l'exécution en appelant la fonction *Sa*. Son but est de coordonner les appels aux fonctions chargées de compiler le terme. Les raisons pour lesquelles les paramètres de cette fonction sont respectivement initialisés à 0, 1 et 'q' sont les suivantes :

- nous commençons à lire le terme. Cela revient donc à parcourir le tableau *s* de la gauche vers la droite. La borne de gauche de ce tableau vaut donc 0;
- *regind* représente l'indice du registre associé au terme;
- le dernier paramètre, *c*, est utilisé comme flag. Il permet de déterminer si la compilation du terme sera exécutée selon un atome question ( $c$ = 'q' ) ou un atome programme ( $c$ = 'p' ).

L'exécution de *Sa* débute. Dans un premier temps, nous sauvons le pointeur vers la table des foncteurs; ceci afin de pouvoir retrouver le nom du terme. Ensuite, nous lançons la lecture du terme en appelant la fonction *lire_sous_terme*. Cette fonction va nous permettre de connaître l'arité du terme en cours de décomposition. Pour notre exemple, elle vaudra trois. Cette fonction va également aller modifier les tableaux *tabvar*, *tabfonct* ainsi que les variables *s*, *ch*, *index* et *itf*. Enfin, elle va construire un début de tokenization du terme en modifiant le tableau *listeS*. Nous laissons au lecteur le soin d'exécuter cette fonction s'il le désire. Toujours est-il qu'au terme de cet appel, nous avons nos variables dans l'état suivant :

**itf** = 2
**ch** = ')'
**index** = 5
**lettre** = 19
**util** et **s** inchangés

**tabfonct**

| nom |
|---|
| 'p' |
| 0 |
| ... |
| 0 |

**tabvar**

| nom | reg |
|---|---|
| 'Y' | 4 |
| 0 | 0 |
| ... | ... |
| 0 | 0 |

**listeS**

| type | reg | pos |
|---|---|---|
| 1 | 2 | 2 |
| 1 | 3 | 7 |
| 0 | 4 | ? |
| ... | ... | ... |

Nous pouvons ici nous lancer dans l'interprétation du tableau *listeS*. Il suffit de respecter les règles suivantes :

1. Si *listeS[i].type* = 1, le champ *pos* est défini. Il contient l'indice du début d'une sous-chaîne dans *s*. Cette sous-chaîne représente le terme associé au registre *listeS[i].reg*.

2. Si *listeS[i].type* = 0, cela signifie que le terme associé au registre *listeS[i].reg* est une variable. Il suffit de connaître son nom. Celui-ci se trouve dans *tabvar[j].nom* où $j$ est tq *listeS[i].reg* = *tabvar[j].reg*.

En appliquant ces deux règles, nous constatons que nous avons le début de tokenization suivant :

$$\begin{aligned} X_2 &= \text{f(X)} \\ X_3 &= \text{h(Y,f(a))} \\ X_4 &= \text{Y} \end{aligned}$$

Ayant à compiler le terme sous forme d'une question, il est nécessaire de poursuivre la tokenization[1]. Pour cela, il suffit de remarquer que nous sommes de nouveau devant le cas qui nous préoccupait au départ. En effet, il nous faut maintenant tokenizer les deux termes f(X) et h(Y,f(a)). Une manière de le faire est de rappeler recursivement *Sa* en donnant comme paramètres le début de chaque chaîne, le registre associé au terme et le flag de compilation. Il se fait que la valeur de ces deux premiers paramètres sont contenus respectivement dans *listeS[i].pos* et *listeS[i].reg*, $\forall i$ tq *listeS[i].type* = 1.

Effectuons le premier appel recursif avec *listeS[i].pos* = 2, *listeS[i].reg* = 2 et $c$ = 'q'. Juste avant le retour de cet appel, nous trouverons nos variables dans l'état suivant :

**ch** = ')'
**itf** = 3
**index** = 6
**lettre** = 6
**util** et **s** inchangés

**tabvar**

| nom | reg |
|---|---|
| 'Y' | 4 |
| 'X' | 5 |
| 0 | 0 |
| ... | ... |
| 0 | 0 |

**tabfonct**

| nom |
|---|
| 'p' |
| 'f' |
| 0 |
| ... |
| 0 |

**listeS'**

| type | reg | pos |
|---|---|---|
| 0 | 5 | ? |
| ... | ... | ... |

Et nous pouvons interpréter *listeS'* comme :

$$X_5 = \text{X}$$

N'ayant plus de structures à tokenizer, nous pouvons sortir de l'appel récursif, sans toutefois oublier de compiler cette première partie de terme. Cette compilation va aller

[1]la tokenization partielle peut déjà suffire pour commencer à compiler le terme sous la forme d'un programme

modifier le tableau *util* ( *util[5]* = 1). En effet, le registre 5 vient d'être alloué à la variable de nom X. Si à l'avenir, une nouvelle occurrence d'une variable qui porte le même nom survient, il suffira de recopier la valeur du registre pour compiler la variable.

Nous voici donc au deuxième appel récursif. Nous laisserons au lecteur le soin de l'effectuer. Toutefois nous pouvons déjà nous douter qu'il y aura un troisième appel. En effet, dans h(Y,f(a)), il nous faudra encore décomposer f(a). Au terme de l'exercice, il sera facile de constater que la tokenization du terme de départ a bien été construite.

### 5.1.4 Exécution des instructions WAM

Le langage $L_0$ ne permettant qu'une unification entre deux termes, l'exécution des instructions WAM, issues de la compilation des termes, est purement séquentielle. C'est pourquoi, l'exécution des instructions Wam se fait en parallèle avec la compilation du terme. Ainsi, quand nous avons interprété une partie du terme et déterminé l'instruction WAM, celle-ci est exécutée.

## 5.2 L'analyseur de WAM 1

Nous allons maintenant nous attacher à l'analyseur de la machine $\langle L_1, M_1 \rangle$. Dans un premier temps, nous étendrons nos conventions. Ensuite, nous décrirons les structures de données ajoutées à celles de l'analyseur précédent. Notons que nous ferons référence à ce que nous avons dit pour l'analyseur de WAM 0 si aucune modification n'a été apportée. Dans un troisième point, nous aborderons l'idée de base de notre algorithme. Enfin nous terminerons sur un commentaire relatif à l'exécution des instructions WAM dans cette version du programme.

### 5.2.1 Conventions

En plus des conventions que nous avons déjà prises dans WAM0, nous ajoutons que le nombre maximum de faits à compiler ne doit pas dépasser *nb_tete* ( =100 ). L'utilisateur peut bien entendu changer cette nouvelle convention en modifiant la valeur de cette constante dans le fichier ANVAR.H de WAM1.

Nous conviendrons aussi de parler d'"atome" pour désigner le terme dont le foncteur est un prédicat. Nous utiliserons le mot "terme" pour désigner l'argument d'un prédicat. Ainsi, si nous avons le fait p(f(X),h(Y,f(a)),Y), l'atome est p(f(X),h(Y,f(a)),Y) et ses arguments sont les termes f(X), h(Y,f(a)) et Y.

### 5.2.2 Structures de données

Les structures de données sont entièrement décrites dans le fichier ANVAR.H. Les différences à noter par rapport à celles de l'analyseur précédent sont :

- l'ajout de deux nouveaux tableaux *label* et *tablisteS*;
- les modifications des tableaux *tabvar*, *tabfonct* et *listeS*.

Nous commençons par la description des nouveaux tableaux avant d'aborder les modifications qui ont affectés les tableaux des variables et des foncteurs.

**label** : Dans cette machine, nous avons introduit l'instruction WAM de saut "Call @p/n" où @p/n est une adresse. Le tableau *label* a pour but de gérer ces sauts. Il est composé de deux champs : *etiqt* et *indice*. Le premier contient le nom du prédicat qu'il faut référencer. Le second contient la valeur de @p/n i.e. *indice* contient l'adresse où se poursuivra l'exécution des instructions WAM.

**tablisteS** : Il s'agit d'un tableau dont le seul champ est le tableau *listeS*. Sa raison d'être est purement technique. Il permet de garder une copie des tableaux *listeS* lors de la compilation des arguments d'un prédicat.

**tabvar** : Nous ajoutons à ce tableau un nouveau champ $A$. Nous l'appellerons parfois le registre argument. Il contiendra le numéro d'un registre associé à un argument si cet argument est une variable. Il vaudra 0 sinon.

**tabfonct** : Un nouveau champ lui est aussi ajouté. Il s'agit de *arite*. La cellule *tabfonct[i].arite* contiendra l'arité du foncteur de nom *tabfonct[i].nom*.

**listeS** : Il ne s'agit pas de la modification physique de la variable; mais de ce qu'elle va maintenant représenter. Lorsque *listeS[i].pos=0*, cela signifie que le $i^{eme}$ argument du terme que nous décomposons est une variable. Le champ *listeS[i].pos* contiendra dorénavant un pointeur. Celui-ci est l'indice de la cellule du tableau *tabvar* qui contient le nom de cette variable.

### 5.2.3 Idée de l'algorithme

Nous devons assurer dans $M_1$ la compilation de plusieurs faits. Ceux-ci étant des atomes, le problème se ramène à répéter un certain nombre de fois la compilation d'un atome. Mais compiler un atome revient à compiler ses arguments qui sont des termes. Or nous savons déjà compiler des termes. Pour que la compilation d'un atome puisse se ramener à ce cas, il suffira lors d'une première lecture de trouver ces termes. Nous pourrons dès lors leur associer un registre $A_i$. Si le fait doit être compilé sous la forme d'une question, il sera nécessaire de sauver la représentation de ces termes dans le tableau *tablisteS*. Ensuite, on peut reprendre la représentation de ces termes et effectuer leur compilation comme dans $M_0$.

Signalons que la première étape de cette compilation est assurée par la fonction *lire_terme*. La seconde étape est assurée par *Sa*. L'idée générale est fort simple. Il y a bien sûr eu quelques problèmes techniques qu'il a fallu surmonter comme par exemple la gestion des adresses lors des appels dûs à l'introduction de la nouvelle instruction "Call". Il a également fallu assurer une bonne gestion des registres arguments qui étaient alloués. Toutefois ces problèmes ne sont pas difficiles à résoudre. C'est pourquoi, nous ne les détaillerons pas ici. Nous laisserons le lecteur interessé chercher la réponse dans le fichier ANALYSE.CPP.

### 5.2.4 L'exécution

Un programme de $M_0$ se compose maintenant d'un ensemble de faits. Nous ne savons pas à l'avance avec quelle question l'utilisateur va exécuter le programme. Il n'est donc plus

possible de compiler et d'exécuter le programme en même temps. Il nous faut donc d'abord compiler le programme, ensuite engendrer un code[2] et le stocker. Ce stockage a lieu dans la zone *Code* du tableau *Mem*. Lorsque l'utilisateur pose sa question, nous la compilons, nous engendrons un code et nous le stockons. L'exécution commence en interprétant le code de cette question puis celui du programme.

En résumé, nous pouvons dire maintenant que le programme se compose de deux parties distinctes à savoir :

1. L'analyse, la compilation et le stockage du code généré. Ces opérations peuvent se faire simultanément.

2. L'exécution de ce code.

Nous trouverons cette approche dans les autres machines. Nous ne reviendrons donc plus à l'avenir sur ce point.

[2]La génération d'un code permet de sauver un maximum d'information dans un minimum de place

## 5.3 L'analyseur de WAM 2

Nous allons ici aborder l'analyseur de la machine $\langle L_2, M_2 \rangle$. Comme dans les parties précédentes, nous commencerons par les nouvelles conventions que nécessitent ce programme. Ensuite, nous décrirons les nouvelles variables que nous avons introduites et nous terminerons par l'idée générale de l'algorithme.

### 5.3.1 Conventions

Nous avons introduit deux nouvelles constantes, à savoir :

- max_alloc qui est le nombre maximum de clauses que nous pouvons avoir dans le programme;
- ref_max qui est le nombre maximum de références aux appels que le programme est capable de traiter.

Le lecteur peut encore changer les valeurs de ces constantes. Elles se trouvent dans le fichier ANVAR.H du répertoire WAM2. Toutefois, nous devons lui rappeler qu'il lui est interdit de créer des tableaux statiques dont la place mémoire dépasse 64K [3]. Cependant, s'il désire quand même franchir cette limite, une solution consiste à redéclarer ces tableaux comme étant des variables dynamiques. Il est évident qu'il s'agit là d'un choix du moindre mal et qu'il y a certainement d'autres solutions; mais nous laissons cela à l'appréciation et à l'analyse que peut en faire le lecteur.

### 5.3.2 Structures de données

La définition de ces structures en C se trouve détaillée dans le fichier ANVAR.H de WAM 2. Notons ici que la totalité des variables de WAM 1 se retrouve dans WAM 2. Nous n'avons donc fait qu'ajouter des variables. En voici la liste :

**tab_alloc** il s'agit d'un tableau comprenant les champs *nom* et *vu*. Le champ *nom* est un tableau de caractères qui contient le nom des variables qui se trouvent dans un fait ou dans une clause. Le champ *vu* est un flag. Il est égal à un si la variable dont le nom se trouvant dans le premier champ est une variable permanente. Ce flag vaut sinon zéro.

[3] Contrainte due au système d'exploitation DOS.

**tab_ref_avant** Dans cette version de la machine, nous pouvons avoir des clauses dont le corps n'est pas vide. Comme nous le décrirons plus loin, nous lisons les clauses dans l'ordre dans lequel elles sont introduites. Il se peut donc que le corps de la clause contienne des appels à des noms de prédicats qui n'ont pas encore été rencontrés dans une tête de clause lors de la lecture. Nous devons donc retenir les informations concernant ces appels. Nous les conserverons dans ce tableau. Pour ce faire, ce tableau est constitué des champs *nom*, *art*, *p* et *etiq*. Le champ *nom* est un pointeur vers la cellule de *tabfonct* qui contient le nom du prédicat appelé. Le champ *art* contient l'arité de ce prédicat. L'adresse du mot mémoire qui devra contenir la référence à la clause dont le prédicat de tête n'a pas été rencontré est sauvée dans le champ *P*.

**fait** Il s'agit d'un flag qui vaut un si la clause que nous compilons est un fait. Il vaut sinon zéro.

**N** Il contient le nombre de variables permanentes dans une clause.

**TA** C'est un pointeur vers la première cellule libre du tableau des allocations *tab_alloc*.

**RA** C'est un pointeur vers la première cellule libre du tableau des références en avant *tab_ref_avant*.

**indexy** Cette variable conserve le numéro du premier registre libre qui peut contenir une variable permanente.

**index_max** et **art_max** sont deux variables techniques.

### 5.3.3 Idée de l'algorithme

Il s'agit de compiler les clauses au fur et à mesure qu'elles sont introduites au clavier par le programmeur. Nous allons essayer d'effectuer cette compilation en nous ramenant à celle des atomes. Pour ce faire, il suffit de remarquer que si la clause se ramène à un fait, l'algorithme utilisé dans la machine WAM 1 est suffisant. Toutefois pour savoir si c'est un fait, il est nécessaire de lire une première fois la totalité de la clause. Si le symbole ":-" n'est pas aperçu, le corps de la clause est vide; il s'agit donc d'un fait. Nous profiterons aussi de cette première lecture pour compter le nombre de variables libres et permanentes que la clause contient, s'il y en a. Nous pourrons également connaître l'arité de la tête de clause et celle de l'ensemble ses buts. Signalons que *art_max* contiendra la plus grande de ces arités, ceci afin de savoir quel sera le premier registre qui sera certainement libre. Connaissant ces paramètres, nous pouvons appliquer l'algorithme de WAM 1 à la clause. En effet, la tête de clause peut se compiler comme un fait de WAM 1. Chaque but du

corps de la clause peut se compiler comme une question de WAM 1. Nous venons ainsi de ramener la compilation d'une clause au programme précédent.

Il faut bien sûr être attentif aux variables temporaires et permanentes. Il faudra également veiller à tenir à jour les références en avant. Mais il s'agit là de problèmes techniques que nous ne détaillerons pas. Nous préférons renvoyer le lecteur qui veut avoir plus d'informations concernant ces problèmes au programme ANALYSE.CPP de WAM 2.

En résumé, cet algorithme fonctionne en deux passes :

1. La première passe détermine l'ensemble des informations nécessaires pour la compilation;
2. la deuxième compile la clause en la découpant en atomes et en tenant compte des paramètres fournis par la première passe.

Ensuite l'exécution du programme compilé sera lancée en commençant, comme c'était le cas pour WAM 1, par la question.

## 5.4 L'analyseur de WAM 3

Il nous reste maintenant à voir le dernier analyseur. Le contenu de ce qui suit, s'articulera de la même manière que les autres parties. Nous commencerons donc par introduire les nouvelles constantes suivies de la description des nouvelles variables. Dans la troisième partie, nous expliquerons comment nous avons géré le backtracking.

### 5.4.1 Conventions

Nous supposerons dans cette version de l'analyseur que le programme Prolog se trouve dans un fichier. Le nom de ce fichier, chemin et extension compris ne peut pas dépasser *nom_max*(=20) caractères. Nous supposerons également que le nombre maximum de noms de tête de clauses différents (nom de procédures Prolog) ne dépasse pas *max_tête* (=100). Enfin, il nous sera interdit d'avoir un programme avec plus de *max_clause*(=30) faits ou clauses.

Cette dernière limitation est assez contraignante. Elle vient du fait qu'un tableau déclaré statiquement ne peut pas dépasser 64K. Pour contourner ce problème, il suffira de déclarer ce tableau comme une variable dynamique. Une autre solution serait de gérer cette structure dans un fichier. Nous laisserons le soin de choisir au lecteur qui désire modifier ces conventions. Nous lui rappelons qu'il les trouvera dans le fichier ANVAR.H de WAM 3.

### 5.4.2 Structures de données

Comme pour la deuxième machine, WAM 3 contient la totalité des variables de WAM 2. Seules cinq structures ont été ajoutées. Les voici :

**prg** Il s'agit de la variable fichier qui contiendra le programme Prolog.

**nom** C'est la chaîne de caractères qui contiendra le nom du fichier.

**tab_tete** Il s'agit d'un tableau composé de quatre champs. Comme un programme Prolog peut être constitué de clauses ayant le même nom de tête, ce tableau va permettre de gérer les appel à ces clauses. Nous verrons comment sur un exemple dans la partie qui suit. Pour le moment, il est suffisant de savoir que les champs *nom* et *art* contiendront respectivement le nom et l'arité des prédicats des têtes de clauses. Le champ *nbocc* contient le nombre de clauses qui commencent par le même nom. Le champ *etiq* contient une adresse. Son rôle sera expliqué plus tard.

**TT** C'est un pointeur vers la première cellule libre du tableau *tab_tete*.

**tab_clause** Ce tableau contient l'ensemble des informations permettant de compiler une clause. Il s'agit d'un tableau composé de quatre champs. Le premier *art_max*, contient la plus grande arité de l'ensemble des arités des atomes qui se trouvent dans une clause. Le champ *fait* est un flag qui permet de savoir si la clause est un fait ou non. Le champ *N* contient le nombre de variables permanentes qui se trouvent dans la clause. Enfin, le tableau *variable*, contient dans son champ *nom* le nom des variables qui se trouvent dans la clause. Il permet aussi de savoir si chacune de ces variables est permanente ou temporaire, grâce à son champ *vu*.

### 5.4.3 Idée de l'algorithme

Comme dans la version précédente, nous allons essayer de ramener la compilation des clauses de WAM 3 à celles de WAM 2. Pour ce faire nous traiterons le programme Prolog en deux passes. Lors d'une première passe, nous initialisons les tableaux *tab_tete* et *tab_clause*. Lors de la deuxième passe, nous nous servons de ces informations pour compiler les clauses.

La compilation des clauses peut se faire comme celles de WAM 2. En effet, le tableau *tab_clause* contient l'ensemble des informations utiles pour cela. Le seul problème qui subsiste est l'appel aux clauses lorsque plusieurs choix sont possibles, c'est-à-dire lorsque plusieurs clauses commencent par le même prédicat. Le tableau *tab_tete* va nous permettre de remédier à cela. Voyons comment sur un exemple. Supposons que nous ayons le programme suivant :

$$p(a, X).$$
$$p(X, b).$$
$$p(X, Y) :- \quad p(X, Z), p(Z, Y).$$

Lors de la première passe, le tableau *tab_tete* sera initialisé comme suit :

**nom** = 'p'
**art** = 2
**nbocc** = 2
**etiq** = 0

Nous commençons la compilation du premier fait. Comme le champ *nbocc* $\neq 0$, cela signifie qu'il y a encore d'autres points de choix; deux dans notre exemple. Nous pouvons

dès lors signaler ce fait en compilant l'instruction WAM "Try me else $P$" où $P$ est une adresse qui référence le prédicat suivant. Plus précisément c'est l'adresse dans le tableau *Code* où se trouve le début du code compilé du deuxième fait. Cette adresse nous est encore inconnue. Il nous faut donc mémoriser l'adresse où nous devrons revenir mettre à jour cette référence au prédicat suivant. Nous la stockerons dans le champ *etiq*. Dans cet exemple, *etiq* contiendra l'adresse 0. Puisque nous avons un point de choix en moins, Il ne faut pas oublier de décroître d'une unité *nbocc* avant de poursuivre.

Nous continuons donc la compilation du fait comme nous l'aurions exécutée dans WAM 2 et nous arrivons au deuxième fait. Nous connaissons l'adresse $P'$ où nous sommes arrivés dans la compilation. Nous pouvons donc mettre à jour la référence $P$ qui se trouve à l'adresse *etiq* (*Code[etiq]=P'*). Cela étant fait, *nbocc* vaut 1. Cela signifie que nous avons encore un point de choix. Nous devons donc compiler l'instruction WAM "Retry me else $P$" " où $P$" est la référence vers le dernier point de choix. Nous ne la connaissons toujours pas. Nous sauvons donc dans *etiq* l'adresse où nous devrons mettre cette référence. Et nous n'oublions pas de diminuer le compteur *nbocc* de un avant de continuer de compiler ce fait.

Nous arrivons dès lors à la troisième clause, dernier point de choix pour le prédicat '*p*'. Nous pouvons mettre à jour la référence $P$" et commencer la compilation de la clause par l'instruction " Trust me" puisque *nbocc* vaut 0. Le reste de la compilation de la clause est semblable à celle de WAM 2. Rappelons ici que les trois instructions que nous avons utilisées permettent la gestion du backtracking.

Nous venons donc de faire le tour des différents analyseurs que nous avons conçus, et nous avons vu que chacun d'entre eux se sert du précédent pour résoudre le problème de la compilation. Il reste peut-être au lecteur des points nébuleux. Nous lui conseillons, si c'est le cas, de se reporter aux analyseurs (fichiers ANALYSE.CPP) qui se trouvent en annexe ou sur disquette. Avec un peu de patience et de persévérance, nous sommes persuadés qu'il trouvera la réponse à ses questions.

## 5.5 Guide d'utilisation des interpéteurs WAM

Nous décrivons le contenu de la disquette ci-jointe. En plus, bien que l'utilisation de ces programmes soit simple, nous en donnons un mode d'emploi.

### 5.5.1 Description

La disquette accompagnant ce travail se divise en quatre répertoires intitulés WAM0, WAM1, WAM2 et WAM3 correspondant bien entendu aux quatre étapes de la machine abstraite de Warren.

Chaque répertoire WAMi ( $0 \leq i \leq 3$ ) contient les fichiers suivants :

ANALYSE.CPP
INSTRWi.CPP
ANVAR.H
CVAR.H

Le fichier ANALYSE.CPP contient le code source de l'analyseur syntaxique spécifique à la machine. Le fichier INSTRWi.CPP contient le code source des instructions spécifiques à la machine. Les fichiers ANVAR.H et CVAR.H contiennent respectivement les déclarations des constantes, des types et des variables de l'analyseur et des instructions. Signalons que le répertoire WAM3 contient en plus des exemples de programmes Prolog purs : A.PRO, APPEND.PRO et POUSSY.PRO.

### 5.5.2 Mode d'emploi

#### a. Lancement

Le nom du fichier exécutable est pour chaque machine "analyse". Nous proposons deux versions de chaque machine : l'une avec occur-check et l'autre sans. Si l'utilisateur souhaite utiliser la version incluant l'occur-check, il doit ajouter le switch "o" précédé d'un espace à la commande "analyse". Dans tous les autres cas, l'exécution se fera sans occur_check. Par exemple, une exécution avec occur_check se fera en tapant

```
a:\WAM0>analyse o
```

#### b. Déroulement

Les trois premières versions sont interactives. Chaque interpéteur demande d'abord la ou les lignes de programme. Puis, quand l'utilisateur souhaite passer à la question, il lui suffit de la faire précéder de "?-" comme spécifié au chapitre 3. Dans la version complète WAM3,

nous invitons l'utilisateur à fournir un fichier contenant uniquement son programme Prolog. Le système lui demandera le nom de ce fichier. Si ce fichier ne se trouve pas dans le répertoire courant, une erreur surviendra. Il est donc préférable de spécifier en même temps que le nom du fichier son chemin. Il pourra ensuite poser sa question à l'invite du programme. Nous présentons alors le contenu du programme Prolog, le code WAM généré et les solutions proposées.

Nous prions le lecteur de nous excuser si de nombreux problèmes existent encore au niveau de l'interface. Nous avons préféré n'en fournir qu'une ébauche au profit des interpéteurs et de la rédaction de ce travail.

# Bibliographie


[AK91] H. Aït-Kan. *Warren's abstract machine, a tutorial reconstruction.* The MIT press, 1991.

[Boi88] P. Boizumault. *Prolog : l'implantation.* Etudes et recherches en informatique. Masson, 1988.

[LC93] B. Le-Charlier. Computational logic. Notes de cours, FUNDP, 1993.

[LC94] B. Le-Charlier. L'analyse statique du programme par interprétation abstraite. *Nouvelles de la Science et des Technologies*, 9(4):19–25, 1994.

[LCVH94] B. Le-Charlier and P. Van-Hentenryck. Experimental evaluation of a generic abstract interpretation algorithm for prolog. *ACM Transactions on Programming Languages and Systems ( TOPLAS )*, 16(1):35–101, January 1994.

[Llo87] J.W. Lloyd. *Foundations of logic programing.* Symbolic computation, Artificial intelligence. Springer Series, second extended edition edition, 1987.

[Sch94] P-Y. Schobbens. Techniques d'intelligence artificielle. Notes de cours, FUNDP, 1994.

[SS86] L. Sterling and E. Shapiro. *The art of Prolog.* Advanced programming techniques. The MIT press, 1986.

# ANVAR.H

```
#define lg_var_max 25
#define lg_fonct_max 20
#define lg_str_max 80
#define l_max 20
#define nb_var 50
#define nb_fonct 100
#define arite_max 20
#define ref_max 40
#define nb_tete 100
#define max_alloc 40
# define max_clause 30
# define max_tete 100
# define nom_max 20

typedef struct
        {unsigned pos:8;
         unsigned reg:5;
         unsigned type:3;
        }tlisteS;

typedef struct
        {tlisteS l[l_max];
        }ttablisteS;

typedef struct
        {char nom[lg_var_max];
         unsigned reg;
         unsigned A;
        }ttabvar;

typedef struct
        {char nom[lg_fonct_max];
         unsigned arite;
        }ttabfonct;

typedef struct
        {unsigned indice;
         unsigned etiqt;
         char type;
        }tlabel;

typedef struct
 {
  unsigned nom ;
  unsigned art ;
  unsigned p ;
  unsigned etiq ;
 } ttab_ref_avant ;

typedef struct
 {
  char nom[lg_var_max] ;
  unsigned vu ;
 } ttab_alloc ;

 typedef struct
```

```
 {
  unsigned art_max ;
  unsigned fait ;
  unsigned N ;
  ttab_alloc variables[max_alloc] ;
 } ttab_clause ;

typedef struct
 {
  char nom[lg_fonct_max] ;
  unsigned art ;
  unsigned nbocc ;
  int etiq ;
 } ttab_tete ;


char tc,tf,ch[lg_var_max],s[lg_str_max],tete[lg_fonct_max],
    question[lg_str_max],nom[nom_max];
int lettre,index,indexy,index_max;
ttab_clause tab_clause[max_clause] ;
ttab_tete tab_tete[max_tete] ;
ttabvar tabvar[nb_var];
ttabfonct tabfonct[nb_fonct];
tlabel label[nb_tete];
ttab_ref_avant tab_ref_avant[ref_max] ;
int util[arite_max];
unsigned ilbl,itf,TA,arite,RA;
unsigned TT,i ;
int px1,px2,py1,py2 ;
int cx1,cx2,cy1,cy2 ;
int fx,fy ;
int qx,qy ;

FILE *prg;
```

# CVAR.H

```
# define registre_max 21
# define heap_max 1000+registre_max
# define stack_max 500+heap_max
# define trail_max 500+stack_max
# define code_max 500+trail_max
# define pile_max 100
# define mem_max code_max

typedef struct {
                    unsigned art;
                    unsigned nom;
                }tcel;

union tcellule {
                    unsigned long mot;
                    tcel cel;
                };

union tmemoire{
                unsigned long Store[mem_max] ;
                tcellule X[mem_max] ;
                tcellule Heap[mem_max] ;
                tcellule Stack[mem_max] ;
                tcellule Trail[mem_max] ;
                tcellule Code[mem_max] ;
               };
tmemoire Mem;
unsigned long H,P,P_debut,S,E,CP,ind_arg,N,Num_of_args,HB,
              B,TR;
char mode;
tcellule Argument[arite_max];
```

```
/* declaration des fonctions de travail */

unsigned long type(unsigned long nbre);
unsigned long adresse(unsigned long nbre);
unsigned long gauche_dcl(unsigned long mot,char bit);
unsigned long droite_dcl(unsigned long mot,char bit);
unsigned long ch_arite(unsigned long mot);
unsigned long ch_fonct(unsigned long mot);

/* definition des foncions de travail */

unsigned long type(unsigned long nbre)
{
 nbre=nbre << 29;
 nbre=nbre >> 29;
 return(nbre);
}

unsigned long adresse(unsigned long nbre)
{
 nbre=nbre >> 3;
 return(nbre);
}

unsigned long gauche_dcl(unsigned long mot,char bit)
{
 mot=mot << bit;
 return(mot);
}

unsigned long droite_dcl(unsigned long mot,char bit)
{
 mot=mot >> bit;
 return(mot);
}

unsigned long ch_arite(unsigned long mot)
{
 unsigned long smot;

 smot=mot;
 mot=mot >> 16;
 mot=mot << 16;
 return(smot-mot);
}

unsigned long ch_fonct(unsigned long mot)
{
 mot=mot >> 16;
 return(mot);
}
```

# INSTRW3.CPP

```
#include <cvar.h>
#include <test.cpp>
/* prototype des fonctions */

void Push(unsigned long valeur,unsigned *sommet,unsigned long PDL[]);
unsigned long Pop(unsigned *sommet,unsigned long PDL[]);
int Empty(unsigned *sommet);
void W3_put_structure(unsigned nom,unsigned arite,unsigned reg);
void W3_set_variable_permanent(unsigned regy);
void W3_set_variable_temporary(unsigned regx);
void W3_set_value_permanent(unsigned regy);
void W3_set_value_temporary(unsigned regx);
char occur_check(unsigned long a1,unsigned long a2);
unsigned long Deref(unsigned long adresse);
void W3_Bind_occur(unsigned long a1,unsigned long a2);
void W3_Bind(unsigned long a1,unsigned long a2);
char W3_Unify_occur(unsigned long a1,unsigned long a2);
char W3_Unify(unsigned long a1,unsigned long a2);
char W3_get_structure_occur(unsigned nom,unsigned arite,unsigned reg);
char W3_get_structure(unsigned nom,unsigned arite,unsigned reg);
void W3_unify_variable_permanent(unsigned regy);
void W3_unify_variable_temporary(unsigned regx);
char W3_unify_value_permanent_occur(unsigned regy);
char W3_unify_value_permanent(unsigned regy);
char W3_unify_value_temporary_occur(unsigned regx);
char W3_unify_value_temporary(unsigned regx);
void W3_put_variable_permanent(unsigned regy,unsigned rega);
void W3_put_variable_temporary(unsigned regx,unsigned rega);
void W3_put_value_permanent(unsigned regy,unsigned rega);
void W3_put_value_temporary(unsigned regx,unsigned rega);
void W3_get_variable_permanent(unsigned regy,unsigned rega);
void W3_get_variable_temporary(unsigned regx,unsigned rega);
char W3_get_value_permanent_occur(unsigned regy,unsigned rega);
char W3_get_value_permanent(unsigned regy,unsigned rega);
char W3_get_value_temporary_occur(unsigned regx,unsigned rega);
char W3_get_value_temporary(unsigned regx,unsigned rega);
char W3_call(unsigned nom);
void W3_proceed();
void W3_allocate(unsigned n) ;
void W3_deallocate() ;
void W3_try_me_else(unsigned L) ;
void W3_retry_me_else(unsigned L) ;
void W3_trust_me() ;
void W3_unwind_trail( long a1, long a2 ) ;
void W3_trail(unsigned long a) ;
char backtrack() ;

/* declaration des fonctions */

void Push(unsigned long valeur,unsigned *sommet,unsigned long PDL[])
{
 *sommet=*sommet+1;
 PDL[*sommet]=valeur;
}


unsigned long Pop(unsigned *sommet,unsigned long PDL[])
```

```
{
 *sommet=*sommet-1;
 return(PDL[*sommet+1]);
}


int Empty(unsigned *sommet)
{
 if (*sommet==0)
   return(1);
 else
   return(0);
}

void W3_put_structure(unsigned nom,unsigned arite,unsigned reg)
{
 tcellule cellule;
 cellule.cel.nom=nom;
 cellule.cel.art=arite;
 Mem.Heap[H].mot=(H+1)*8+1;
 Mem.Heap[H+1]=cellule;
 Mem.X[reg].mot=Mem.Heap[H].mot;
 H=H+2;
}

void W3_set_variable_permanent(unsigned regy)
{
 Mem.Heap[H].mot=H*8+0;
 Mem.Stack[E+2+regy].mot=Mem.Heap[H].mot;
 H++;
}

void W3_set_variable_temporary(unsigned regx)
{
 Mem.Heap[H].mot=H*8+0;
 Mem.X[regx].mot=Mem.Heap[H].mot;
 H++;
}

void W3_set_value_permanent(unsigned regy)
{
 Mem.Heap[H].mot=Mem.Stack[E+2+regy].mot;
 H++;
}

void W3_set_value_temporary(unsigned regx)
{
 Mem.Heap[H].mot=Mem.X[regx].mot;
 H++;
}

char occur_check(unsigned long a1,unsigned long a2)
{
 int i;
 unsigned arite;
 unsigned long l;
 char echec;

 echec=0;
```

```
 i=1;
 if (type(Mem.Heap[a2].mot) != 0)
 { l=adresse(Mem.Heap[a2].mot);
   arite=Mem.Heap[l].cel.art;
   while ((i != arite+1)&&(echec == 0))
   { if (type(Mem.Heap[l+i].mot) == 0)
     { if (adresse(Mem.Heap[l+i].mot) == adresse(Mem.Heap[a1].mot))
            echec=1;
     }
     else
          echec=occur_check(a1,l+i);
     i++;
   }
 }
 return(echec);
}

unsigned long Deref(unsigned long adr)
{
 unsigned long cel;

 cel=Mem.Store[adr];
 if ((type(cel)==0)&&(adresse(cel)!=adr))
   return(Deref(adresse(cel)));
 else
   return(adr);
}

void W3_Bind_occur(unsigned long a1,unsigned long a2)
{
 char echec,t1,t2;

 t1 = type ( Mem.Store[a1] );
 t2 = type ( Mem.Store[a2] );
 if ((t1==0)&&((t2!=0)||(a2<a1)))
 { if (mode == 'r')
   { echec=occur_check(a1,a2);
     if (echec != 0)
     { puts("! occur check !");
       exit(1);
     }
   }
   Mem.Store[a1]=Mem.Store[a2];
   W3_trail(a1) ;
 }
 else
 { echec=occur_check(a2,a1);
   if (echec != 0)
   { puts("! occur check !");
     exit(1);
   }
 }
 Mem.Store[a2]=Mem.Store[a1];
 W3_trail(a2) ;
}

void W3_Bind(unsigned long a1,unsigned long a2)
{
 char echec,t1,t2;
```

```
t1 = type ( Mem.Store[a1] );
t2 = type ( Mem.Store[a2] );
if ((t1==0)&&((t2!=0)||(a2<a1)))
{ Mem.Store[a1]=Mem.Store[a2];
  W3_trail(a1) ;
}
else
{ Mem.Store[a2]=Mem.Store[a1];
  W3_trail(a2) ;
}
}
```

**char W3_Unify_occur(unsigned long a1,unsigned long a2)**

```
{
tcellule cel1,cel2;
unsigned long d1,d2,valeur,v1,v2,PDL[pile_max];
int echec,i,meme_nom;
unsigned spdl;
char t1,t2;

  spdl=0;
  Push(a1,&spdl,PDL);
  Push(a2,&spdl,PDL);
  echec=0;
  while (!(Empty(&spdl)||echec))
    {d1=Deref(Pop(&spdl,PDL));
     d2=Deref(Pop(&spdl,PDL));
     if (d1!=d2)
           {t1=type(Mem.Heap[d1].mot);
            t2=type(Mem.Heap[d2].mot);
            v1=adresse(Mem.Heap[d1].mot);
            v2=adresse(Mem.Heap[d2].mot);
            if ((t1==0) || (t2==0))
              W3_Bind_occur(d1,d2);
            else
             {
              cel1=Mem.Heap[v1];
              cel2=Mem.Heap[v2];
              meme_nom=strcmp(tabfonct[cel1.cel.nom].nom,tabfonct[cel2.cel.nom].nom);
                if ((meme_nom == 0)&&(cel1.cel.art == cel2.cel.art))
                      {for(i=1;i<=cel1.cel.art;i++)
                       {Push(v1+i,&spdl,PDL);
                        Push(v2+i,&spdl,PDL);
                       }
                      }
                else
                      echec=1;
             };
           };
    };
return(echec) ;
}
```

**char W3_Unify(unsigned long a1,unsigned long a2)**

```
{
tcellule cel1,cel2;
unsigned long d1,d2,valeur,v1,v2,PDL[pile_max];
int echec,i,meme_nom;
```

```
unsigned spdl;
char t1,t2;

 spdl=0;
 Push(a1,&spdl,PDL);
 Push(a2,&spdl,PDL);
 echec=0;
 while (!(Empty(&spdl)||echec))
  {d1=Deref(Pop(&spdl,PDL));
   d2=Deref(Pop(&spdl,PDL));
   if (d1!=d2)
        {t1=type(Mem.Heap[d1].mot);
         t2=type(Mem.Heap[d2].mot);
         v1=adresse(Mem.Heap[d1].mot);
         v2=adresse(Mem.Heap[d2].mot);
         if ((t1==0) || (t2==0))
          W3_Bind(d1,d2);
         else
          {
           cel1=Mem.Heap[v1];
           cel2=Mem.Heap[v2];
           meme_nom=strcmp(tabfonct[cel1.cel.nom].nom,tabfonct[cel2.cel.nom].nom);
             if ((meme_nom == 0)&&(cel1.cel.art == cel2.cel.art))
                  {for(i=1;i<=cel1.cel.art;i++)
                    {Push(v1+i,&spdl,PDL);
                     Push(v2+i,&spdl,PDL);
                    }
                  }
             else
                  echec=1;
          };
        };
  };
 return(echec) ;
}

char W3_get_structure_occur(unsigned nom,unsigned arite,unsigned reg)
{
 char echec;
 unsigned long a,adr;
 tcellule cellule;

 echec=0;
 adr=Deref(reg);
 switch(type(Mem.Store[adr]))
 { case 0 : {
             Mem.Heap[H].mot=(H+1)*8+1;
             cellule.cel.nom=nom;
             cellule.cel.art=arite;
             Mem.Heap[H+1]=cellule;
             W3_Bind_occur(adr,H);
             H=H+2;
             mode='w';
             break;
            }
  case 1 : {
             a=adresse(Mem.Store[adr]);
             cellule=Mem.Heap[a];
             if ((strcmp(tabfonct[cellule.cel.nom].nom,tabfonct[nom].nom)==0)
```

```
                &&(cellule.cel.art==arite))
                { S=a+1;
                      mode='r';
                }
                else
                      echec=1;
                break;
               }
   default : echec=1;
 }
 return(echec);
}

char W3_get_structure(unsigned nom,unsigned arite,unsigned reg)
{
 char echec;
 unsigned long a,adr;
 tcellule cellule;

 echec=0;
 adr=Deref(reg);
 switch(type(Mem.Store[adr]))
 { case 0 : {
                mode='w';
                Mem.Heap[H].mot=(H+1)*8+1;
                cellule.cel.nom=nom;
                cellule.cel.art=arite;
                Mem.Heap[H+1]=cellule;
                W3_Bind(adr,H);
                H=H+2;
                break;
               }
   case 1 : {
                a=adresse(Mem.Store[adr]);
                cellule=Mem.Heap[a];
                if ((strcmp(tabfonct[cellule.cel.nom].nom,tabfonct[nom].nom)==0)
                &&(cellule.cel.art==arite))
                { S=a+1;
                      mode='r';
                }
                else
                      echec=1;
                break;
               }
   default : echec=1;
 }
 return(echec);
}

void W3_unify_variable_permanent(unsigned regy)
{
 switch(mode)
 { case 'r' : Mem.Stack[E+2+regy].mot=Mem.Heap[S].mot;break;
   case 'w' : {
                      Mem.Heap[H].mot=(H*8)+0;
                      Mem.Stack[E+2+regy].mot=Mem.Heap[H].mot;
                      H++;
                      break;
                }
```

```
 }
 S++;
}

void W3_unify_variable_temporary(unsigned regx)
{
 switch(mode)
 { case 'r' : Mem.X[regx].mot=Mem.Heap[S].mot;break;
   case 'w' : {
                    Mem.Heap[H].mot=(H*8)+0;
                    Mem.X[regx].mot=Mem.Heap[H].mot;
                    H++;
                    break;
              }
 }
 S++;
}

char W3_unify_value_permanent_occur(unsigned regy)
{
 char echec ;

 echec=0;
 switch(mode)
 { case 'r' : {
                    echec = W3_Unify_occur(E+2+regy,S);
                    break;
              }
   case 'w' : {
                    Mem.Heap[H].mot=Mem.Stack[E+2+regy].mot;
                    echec=occur_check(H,Deref(H));
                    if (echec == 1)
                    { puts("! occur check !");
                      exit(1);
                    }
                    H++;
                    break;
              }
 }
 S++;
 return(echec);
}

char W3_unify_value_permanent(unsigned regy)
{
 char echec ;

 echec=0;
 switch(mode)
 { case 'r' : {
                    echec = W3_Unify(E+2+regy,S);
                    break;
              }
   case 'w' : {
                    Mem.Heap[H].mot=Mem.Stack[E+2+regy].mot;
                    H++;
                    break;
              }
 }
```

```
 S++;
 return(echec);
}

char W3_unify_value_temporary_occur(unsigned regx)
{
 char echec ;

 echec=0;
 switch(mode)
 { case 'r' : {
                    echec = W3_Unify_occur(regx,S);
                    break;
                }
   case 'w' : {
                    Mem.Heap[H].mot=Mem.X[regx].mot;
                    echec=occur_check(H,Deref(H));
                    if (echec == 1)
                    { puts("! occur check !");
                      exit(1);
                    }
                    H++;
                    break;
                }
 }
 S++;
 return(echec);
}

char W3_unify_value_temporary(unsigned regx)
{
 char echec ;

 echec=0;
 switch(mode)
 { case 'r' : {
                    echec = W3_Unify(regx,S);
                    break;
                }
   case 'w' : {
                    Mem.Heap[H].mot=Mem.X[regx].mot;
                    H++;
                    break;
                }
 }
 S++;
 return(echec);
}

void W3_put_variable_permanent(unsigned regy,unsigned rega)
{
 unsigned long adr ;

 adr = E +regy +2 ;
 Mem.Stack[adr].mot=adr*8+0;
 Mem.X[rega].mot=Mem.Stack[adr].mot;
}
```

```
void W3_put_variable_temporary(unsigned regx,unsigned rega)
{
 Mem.Heap[H].mot=H*8+0;
 Mem.X[regx].mot=Mem.Heap[H].mot;
 Mem.X[rega].mot=Mem.Heap[H].mot;
 H++;
}


void W3_put_value_permanent(unsigned regy,unsigned rega)
{
 Mem.X[rega].mot=Mem.Stack[E+regy+2].mot;
}

void W3_put_value_temporary(unsigned regx,unsigned rega)
{
 Mem.X[rega].mot=Mem.X[regx].mot;
}

void W3_get_variable_permanent(unsigned regy,unsigned rega)
{
 Mem.Stack[E+2+regy].mot=Mem.X[rega].mot;
}

void W3_get_variable_temporary(unsigned regx,unsigned rega)
{
 Mem.X[regx].mot=Mem.X[rega].mot;
}

char W3_get_value_permanent_occur(unsigned regy,unsigned rega)
{
 return(W3_Unify_occur(E+2+regy,rega));
}

char W3_get_value_permanent(unsigned regy,unsigned rega)
{
 return(W3_Unify(E+2+regy,rega));
}

char W3_get_value_temporary_occur(unsigned regx,unsigned rega)
{
 return(W3_Unify_occur(regx,rega));
}

char W3_get_value_temporary(unsigned regx,unsigned rega)
{
 return(W3_Unify(regx,rega));
}

char W3_call(unsigned nom)
{
 char echec ;

 if (nom == 0)
 {
  echec = backtrack() ;
  return( echec ) ;
 }
 else
```

```
{ CP = P + Mem.Code[P].cel.art ;
  Num_of_args =  tabfonct[Mem.Code[P+1].cel.art].arite ;
  P = Mem.Code[P+1].cel.nom ;
  return(0);
 }
}

void W3_proceed()
{
 P = CP ;
}

void W3_allocate ( unsigned n )
{
 unsigned long newE ;

 if(E>B)
   newE = E +Mem.Stack[E+2].mot +3 ;
 else
   newE = B + Mem.Stack[B].mot + 7 ;

 Mem.Stack[newE].mot = E ;
 Mem.Stack[newE+1].mot = CP ;
 Mem.Stack[newE+2].mot = n ;
 E = newE ;
 if (newE+2 >= stack_max)
 { puts("! Memoire STACK depassee !");
   exit(1);
 }
}

void W3_deallocate()
{
 CP = Mem.Stack[E+1].mot ;
 E = Mem.Stack[E].mot ;
}

void W3_try_me_else(unsigned L)
{
 unsigned i;
 unsigned long newB ;

 if (E > B)
   newB=E+Mem.Stack[E+2].mot+3;
 else
   newB=B+Mem.Stack[B].mot+7;

 Mem.Stack[newB].mot=Num_of_args;
 N=Mem.Stack[newB].mot;

 for(i=1;i<=N;i++)
   Mem.Stack[newB+i].mot=Mem.X[i].mot;

 Mem.Stack[newB+N+1].mot=E;
 Mem.Stack[newB+N+2].mot=CP;
 Mem.Stack[newB+N+3].mot=B;
 Mem.Stack[newB+N+4].mot=L;
 Mem.Stack[newB+N+5].mot=TR;
 Mem.Stack[newB+N+6].mot=H;
```

```
 B=newB;
 HB=H;
}

void W3_retry_me_else(unsigned L)
{
 unsigned i ;

 N=Mem.Stack[B].mot;
 for(i=1;i<=N;i++)
  Mem.X[i].mot = Mem.Stack[B+i].mot ;
 E = Mem.Stack[B+N+1].mot ;
 CP = Mem.Stack[B+N+2].mot ;
 Mem.Stack[B+N+4].mot = L ;
 W3_unwind_trail(Mem.Stack[B+N+5].mot,TR) ;
 TR = Mem.Stack[B+N+5].mot ;
 H = Mem.Stack[B+N+6].mot ;
 HB = H ;
}

void W3_trust_me ()
{
 unsigned i ;

 N = Mem.Stack[B].mot ;

 for(i=1;i<=N;i++)
  Mem.X[i].mot = Mem.Stack[B+i].mot ;
 E = Mem.Stack[B+N+1].mot ;
 CP = Mem.Stack[B+N+2].mot ;
 W3_unwind_trail(Mem.Stack[B+N+5].mot,TR) ;
 TR = Mem.Stack[B+N+5].mot ;
 H = Mem.Stack[B+N+6].mot ;
 B = Mem.Stack[B+N+3].mot ;
 HB = Mem.Stack[B+N+6].mot ;
}

void W3_unwind_trail( long a1, long a2)
{
 unsigned i ;

 for(i=a1;i<=(a2-1);i++)
  Mem.Store[Mem.Trail[i].mot] = Mem.Trail[i].mot*8 + 0 ;
}

void W3_trail(unsigned long a)
{
 if ( (a<HB) || ((H<a)&&(a<B)) )
  {
   Mem.Trail[TR].mot=a ;
   TR++ ;
  }
}

char backtrack()
{
 if (B == heap_max)
  return(1) ;
 else
```

```
 {
  P = Mem.Stack[B+Mem.Stack[B].mot+4].mot ;
  return(0) ;
  }
}
```

# ANALYSE.CPP

```
#include<stdio.h>
#include<string.h>
#include<process.h>
#include<stdlib.h>
#include<anvar.h>
#include<IOwam.cpp>
#include<instrw3.cpp>

void premiere_passe();
unsigned long deuxieme_passe(unsigned *itfc);
void lire_clause(char x) ;
unsigned lire_litteral(unsigned x) ;
void lire_arg_litt(unsigned x) ;
void lire() ;
void init_alloc() ;
void maj_alloc(unsigned x,unsigned b) ;
unsigned cherche_var() ;
char test_corps() ;
char test_fin() ;
unsigned calcul_n() ;
void maj_tete(char tete[], unsigned arite) ;
void Sa(unsigned pos,unsigned regind,char c);
unsigned lire_terme(char c);
unsigned lire_sous_terme(unsigned pos,tlisteS listeS[]);
unsigned mettre_var(unsigned reg);
unsigned appartient_tabvar(tlisteS listeS[],unsigned ils);
void compile_debut_quest() ;
void compile_question(unsigned itfc,tlisteS listeS[],unsigned arite,unsigned regind);
void compile_var_quest(char nom[lg_var_max],unsigned *regx,unsigned rega);
void compile_fin_quest(unsigned arite);
void compile_programme(unsigned itfc,tlisteS listeS[],unsigned arite,unsigned regind);
void compile_debut_prgr(unsigned arite);
void compile_var_prgr(char nom[lg_var_max],unsigned *regx,unsigned rega);
void compile_fin_prgr();
char execution_occur();
char execution();
void init_tabvar();
void init_tab_ref_avant();
void init_util();
void init_argument();
void sortie_erreur(int statut);
void decode_terme (tcellule cel) ;
unsigned decode_foncteur (tcellule cel) ;
unsigned permanent(char nom[lg_var_max]);
unsigned compile_clause(char c);
void sauve_contenu_arg();
void maj_references_avant();
void init_label();
void sortie_resultats();
void maj_arguments() ;
void maj_ind_arg();

void main(int argc,char *argv[])
{
 char choix[1];
 unsigned j,itfc;
```

```
if (argc > 2)
 exit(1);

Affiche_Intro() ;
clrscr() ;

choix[0]='o';
while((choix[0] != 'n')&&(choix[0] != 'N'))
{
 fenetre(1,1,79,24) ;
 clrscr() ;
 strcpy(nom,Affiche_Fichier()) ;

 while((choix[0] != 'n')&&(choix[0] != 'N'))
 {
 P=trail_max;
 tabfonct[0].nom[0]='@';
 itf=1;
 ilbl=0;
 RA=0;
 mode='r';
 init_tab_ref_avant();
 init_label();

 Active_Wdw_Prog() ;
 premiere_passe();

 Active_Wdw_Code() ;
 P=deuxieme_passe(&itfc);
 E=heap_max;
 B=heap_max;
 H=registre_max;
 TR=stack_max;
 HB=registre_max;
 ind_arg=0;

 Active_Wdw_Sol() ;
 switch ( strcmp("o",argv[1]) )
 {
  case 0 :{if (execution_occur() != 0)
                cprintf("non, echec.") ;
           else
                sortie_resultats();
           break;
           }
  default:{if (execution() != 0)
                cprintf("non, echec.") ;
           else
                sortie_resultats();
           break;
           }
 }

maj_arguments() ;
gets(choix) ;

 strcpy(choix,Affiche_Boucle_Question()) ;
}
 strcpy(choix,Affiche_Boucle_Prog());
```

```
 }

 Affiche_Bye() ;
 exit(1);
}

void premiere_passe()
{
//---------------------------------------------------------------------
// Cette fonction effectue une premiere passe sur le programme afin d'en
// determiner, pour chaque clause, le foncteur du predicat de tete,
// son arite, l'arite maximale de la clause, l'ensemble des variables
// presentes dans la clause et leur statut permanent ou temporaire.
//
// Uses :
//
// void lire_clause(char x) ;
// void lire() ;
//
//---------------------------------------------------------------------

 char c,fait ;

 prg=fopen(nom,"r+t");
 i = 1 ;
 TT = 0 ;
 fscanf(prg,"%s\n",&s);

 while(strcmp(s,"")!=0)
 {
  cprintf("%s\r\n",s);
  lire_clause('p') ;
  i++ ;
  strcpy(s,"") ;
  fscanf(prg,"%s\n",&s);
 }

//  printf("q : ") ;
//  scanf("%s",&question) ;
 strcpy(question,Affiche_Question()) ;
 strcpy(s,question) ;

 lettre=0 ;
 lire() ;
 lire() ;
 if (ch[0]!='-')
   Affiche_Erreur("mauvaise syntaxe : une question est pr,c,d,e de ?-") ;
 else
   lire_clause('q') ;
 fclose(prg);
}


unsigned long deuxieme_passe(unsigned *itfc)
{
//---------------------------------------------------------------------
// Cette fonction effectue une seconde passe sur le programme afin d'en
// determiner le code d'instructions WAM, y compris l'indice des registres
// et les labels.
```

```
//
// Parametres :
//
// itfc : indice courant dans la table des foncteurs.
// deuxieme_passe : pointeur vers la premiere instruction a executer.
//
// Uses :
//
// unsigned compile_clause(char x) ;
// void lire() ;
// void maj_references_avant() ;
//
//---------------------------------------------------------------------------

 i = 1 ;
 prg=fopen(nom,"r+t");
 fscanf(prg,"%s",&s);
 while(strcmp(s,"")!=0)
 {
  compile_clause('p');

  i++ ;
  strcpy(s,"") ;
  fscanf(prg,"%s",&s);
 }
 fclose(prg);

 strcpy(s,question) ;
 lettre=0 ;
 lire() ;
 lire() ;
 maj_references_avant();

 P_debut=P;
 *itfc=compile_clause('q');
 Mem.Code[P].mot=0;
 return(P_debut);
}

void lire_clause(char x)
{
//---------------------------------------------------------------------------
// Cette fonction lit une clause dans son entierete. Elle calcule l'arite
// maximale de ses litteraux, le nombre de variables permanentes, l'arite
// du predicat de tete et positionne un flag si la clause est ou non un
// fait. Elle tient a jour une table des predicats de tete pour les
// references en avant.
//
// Parametre :
//
// x : flag a 'p' si la clause lue est une ligne de programme et 'q'        //        sinon.
//
// Uses :
//
// void init_alloc() ;
// unsigned lire_litteral(unsigned x) ;
// void maj_tete(unsigned tete,unsigned arite) ;
// char test_corps() ;
// char test_fin() ;
```

```
// void lire() ;
// unsigned calcul_n() ;
//
//-------------------------------------------------------------------------

 unsigned N,art_max,tc,tf ;

 init_alloc() ;

 TA  = 0 ;
 lettre = 0 ;
 tc = 1 ;
 arite = 0 ;
 art_max = 0 ;
 N = 0 ;
 strcpy(tete,"$") ;

 // si x = p donc une ligne de programme
 // alors il faut lire la t•te de la clause .
 if (x=='p')
 {
 // lecture de la t•te de la clause
 arite = lire_litteral(0) ;
 maj_tete(tete,arite) ;
 art_max = arite ;

 // test si le corps est vide
 tc = test_corps () ;
 }
 else
 {
 // x = q donc une ligne de question .
 // lecture de ?-
 lire() ; lire() ;
 }


 // on a un corps soit pour une question soit pour une clause avec ':-'.
 if (tc == 1)
 {
  // lecture du premier but
  arite=lire_litteral(0) ;

  if (art_max < arite)
  {
   art_max = arite ;
  }

  // test si on a parcouru toute la clause
  tf = test_fin() ;
  while (tf==0)
   {
    // lecture des autres buts
    arite=lire_litteral(1) ;
    if (art_max < arite)
    {
     art_max = arite ;
    }
    tf = test_fin() ;
```

```
  }

 N = calcul_n() ;
 tab_clause[i].fait = 0 ;
 }
 else
 {
 tab_clause[i].fait = 1 ;
 }

 tab_clause[i].art_max = art_max ;
 tab_clause[i].N = N ;

}

void lire()
{
//---------------------------------------------------------------------
// Cette fonction lit une chaine de caracteres jusqu'a un caractere
// specifique ou lit un caractere specifique.
// Les caracteres specifiques sont ( , ) : - ? . et \0
// La chaine lue est mise dans le string ch.
//
//---------------------------------------------------------------------

 int i;

 i=0;
 strcpy(ch,"") ;

 while((s[lettre]!='(')&&(s[lettre]!=')')&&(s[lettre]!=':')&&
        (s[lettre]!='?')&&(s[lettre]!='-')&&(s[lettre]!=',')&&
        (s[lettre]!='.')&&(s[lettre]!='\0'))
 {
  ch[i]=s[lettre];
  lettre++;
  i++;
 }
 if (i == 0)
 {
  ch[0]=s[lettre];
  ch[1]='$';
  lettre++;
 }
 else
  ch[i]=0;
}

void maj_alloc(unsigned x,unsigned b)
{
//---------------------------------------------------------------------
// Cette fonction met a jour la table des allocations des variables afin
// de definir si elles sont permanentes ou temporaires.
//
// Parametres :
//
// x : indice du foncteur du litteral lu dans la table des foncteurs.
// b : arite du foncteur en question.
//
```

```
// Uses :
//
// unsigned cherche_var() ;
//
//-------------------------------------------------------------------------

 unsigned k;

 // k est l'endroit o— est (ou sera) la variable rencontr,e
 k = cherche_var() ;

 if(k < b)
  {
   tab_clause[i].variables[k].vu = x ;
  }
 else
  {
   strcpy(tab_clause[i].variables[k].nom,ch) ;
   tab_clause[i].variables[k].vu = 0 ;
  }
}

void init_alloc()
{
//-------------------------------------------------------------------------
// Cette fonction initialise la table d'allocation des variables.
//
//-------------------------------------------------------------------------

 unsigned j ;

 for(j=0;j<max_alloc;j++)
 {
  tab_clause[i].variables[j].nom[0] = '$' ;
  tab_clause[i].variables[j].vu = 0 ;
 }
}

unsigned cherche_var()
{
//-------------------------------------------------------------------------
// Cette fonction renvoit la position de la variable ch dans la table
// d'allocation des variables. Si elle n'y est pas, cette variable sera
// affectee a la premiere cellule libre de la table, qui est pointee par  // TA.
// TA est alors incremente.
//
// Parametre :
//
// cherche_var : position de la variable dans la table d'allocation des
//                       variables.
//
//-------------------------------------------------------------------------

 char trouve ;
 unsigned j ;

 trouve = 0 ;
 j = 0 ;
```

```
 while ((j < TA )&&(trouve==0))
 {
  if(strcmp(tab_clause[i].variables[j].nom,ch)==0)
  {
   trouve = 1 ;
  }
  else
  {
   j++ ;
  }
 }
 // si on n'a pas trouv, alors la var. prend la premiŠre place libre
 if(trouve==0)
 {
  TA ++ ;
 }
 return(j) ;
}

unsigned calcul_n()
{
//---------------------------------------------------------------------
// Cette fonction renvoit le nombre de variables permanentes de la clause
// courante, clause d'indice i, sur base de la table d'allocation des
// variables.
//
// Parametre :
//
// calcul_n : nombre de variables permanentes.
//
//---------------------------------------------------------------------

 unsigned N,j ;

 N=0 ;
 j=0 ;

 while(j<TA )
 {
  if (tab_clause[i].variables[j].vu == 1)
   {
    N = N+1 ;
   }
  j = j+1 ;
 }
 return(N) ;
}

char test_fin()
{
//---------------------------------------------------------------------
// Cette fonction teste si on se trouve en fin de clause.
//
// Parametre :
//
// test_fin : flag a 1 si on est en fin de clause, a 0 sinon.
//
//---------------------------------------------------------------------
```

```
 if ((ch[0]=='.')||(ch[0]=='\0'))
  return(1) ;
 else
 {
  return(0) ;
 }
}

char test_corps()
{
//---------------------------------------------------------------------
// Cette fonction teste si le corps de la clause est vide ou non.
//
// Parametre :
//
// test_corps : flag a 1 si le corps est non vide, a 0 sinon.
//
// Uses :
//
// void Affiche_erreur(char str[80]) ;
// void lire() ;
//
//---------------------------------------------------------------------

 switch(ch[0])
 {
  case ':' : lire() ;
             if(ch[0]=='-')
               return(1) ;
             else
              {
               Affiche_Erreur("mauvaise syntaxe sur :-") ;
               return(0) ;
              }
  case ',' : Affiche_Erreur("dans la clause (,) il manque :- ou ?- ") ;
             return(0) ;
  default : return(0) ;
 }

}

unsigned lire_litteral(unsigned x)
{
//---------------------------------------------------------------------
// Cette fonction lit completement un litteral et renvoit son arite.
//
// Parametre :
//
// x : flag indiquant s'il faut ou non rendre les variables deja      //    rencontrees permanentes ( 1 pour
permanetes ).
// lire_litteral : arite du litteral parcouru.
//
// Uses :
//
// void Affiche_erreur(char str[80]);
// void lire_arg_litt(char x) ;
// void lire() ;
//
//---------------------------------------------------------------------
```

```
unsigned arite ;

lire();
arite = 0 ;
if ((64 < ch[0])&&(ch[0] < 91))
 {
  Affiche_Erreur("une clause doit •tre une construction et non une variable");
   }
else
 {
  strcpy(tete,ch) ;
  lire() ;
  switch (ch[0])
  {
   case '(' : arite = 1 ;
              do
              {
                   lire_arg_litt(x) ;
                   if (ch[0]==',')
                    { arite++ ; }
              }
              while(ch[0]!=')') ;
              lire() ;
              break ;
   case ',' : break ;
   case '.' : break ;
   case ':' : break ;
   default : Affiche_Erreur("mauvaise syntaxe sur les d,limiteurs .ou :");
                    break ;

  }
 }
return(arite) ;
}

void lire_arg_litt(unsigned x)
{
//---------------------------------------------------------------------------
// Cette fonction lit un argument d'un litteral et met a jour la table
// d'allocation des variables.
//
// Parametre :
//
// x : flag indiquant s'il faut ou non rendre les variables deja        //    rencontrees permanentes ( 1 pour
permanetes ).
//
// Uses :
//
// void maj_alloc(char x, unsigned b) ;
// void lire() ;
//
//---------------------------------------------------------------------------

unsigned B,nb ;

B = TA  ;
lire() ;
nb = 1 ;
```

```
while (((ch[0]!=')')||(nb>0))&&((ch[0]!=',')||(nb>1)))
{
 if ((64 < ch[0])&&(ch[0] < 91))
 {
  maj_alloc(x,B) ;
 }
 lire() ;

 switch(ch[0])
 {
  case '(' : nb++ ; break ;
  case ')' : nb--; break ;
 }
}

}


void maj_tete(char tete[],unsigned arite)
{
//-------------------------------------------------------------------------
// Cette fonction maintient a jour la table des predicats de tete afin
// de determiner si plusieurs clauses ont le meme predicat de tete.
// ( but sous-jacent : try_me_else, retry_me_else, trust_me )
//
// Parametres :
//
// tete : foncteur du predicat de tete.
// arite : arite du foncteur.
//
//-------------------------------------------------------------------------

 char trouve ;
 unsigned j ;

 trouve = 0 ;
 j = 0 ;

 while ((j < TT )&&(trouve==0))
 {
  if((strcmp(tab_tete[j].nom,tete)==0)&&(tab_tete[j].art==arite))
  {
   trouve = 1 ;
  }
  else
  {
   j++ ;
  }
 }
 // si on n'a pas trouve, alors la var. prend la premiere place libre
 if(trouve==0)
 {
  tab_tete[TT].nbocc = 1 ;
  tab_tete[TT].etiq = -1 ;
  tab_tete[TT].art = arite ;
  strcpy(tab_tete[TT].nom,tete) ;
  TT++ ;
 }
 else
```

```
 {
  tab_tete[j].nbocc++ ;
 }



}

void Sa(unsigned pos,unsigned regind,char c)
{
//---------------------------------------------------------------------------
//
// Cette fonction coordonne l'appel aux fonctions qui permettent la compi-
// lation des arguments d'une tete de clause, des buts d'une clause ou des
// buts d'une question.
//
// Parametres :
//
// pos : repere dans le tableau s marquant le debut du terme a compiler.
// regind : numero du registre contenant la representation du terme      //          compile.
// c : flag permettant de savoir si l'argument doit etre compile sous la
//     forme d'une question ou d'une tete de clause.
//
// Uses :
//
// unsigned lire_sous_terme(unsigned pos,tlisteS,listeS[]) ;
// void compile_programme(unsigned itfc,tlistes listeS[],unsigned arite,
//                                    unsigned regind) ;
// void compile_question(unsigned itfc,tlisteS listeS[],unsigned arite,
//                                   unsigned regind) ;
// void Sa(unsigned pos,unsigned regind,char c) ;
//
//---------------------------------------------------------------------------

  unsigned itfc;
  tlisteS listeS[l_max];
  unsigned arite,i;

  itfc=itf;
  arite=lire_sous_terme(pos,listeS);
  if (c == 'p')
    compile_programme(itfc,listeS,arite,regind);
  i=0;
  while (i != arite)
  { if ((listeS[i].type == 1)||(listeS[i].type == 2))
      Sa(listeS[i].pos,listeS[i].reg,c);
    i++;
  }
  if (c == 'q')
    compile_question(itfc,listeS,arite,regind);
}

unsigned lire_terme(char c)
{
//---------------------------------------------------------------------------
//
// Cette fonction lit et compile une tete de clause ou un but.
//
// Parametre :
```

```
//
// c : flag permettant de savoir si le terme doit etre compile sous forme
//     d'une question ou d'une tete de clause.
// lire_terme : arite du terme.
//
// Uses :
//
// void lire() ;
// unsigned mettre_var(unsigned reg) ;
// void sortie_erreur(int statut) ;
// void compile_debut_prgr(unsigned arite) ;
// unsigned long compile_var_prgr(char nom[lg_var_max],unsigned *regx,
//                                unsigned rega) ;
// unsigned long compile_var_quest(char nom[lg_var_max],unsigned *regx,
//                                 unsigned rega) ;
// unsigned lire_sous_terme(unsigned pos,tlisteS listeS) ;
// void compile_programme(unsigned itfc,tlistes listeS[],unsigned arite,
//                        unsigned regind) ;
// void Sa(unsigned pos,unsigned regind,char c) ;
// void compile_question(unsigned itfc,tlisteS listeS[],unsigned arite,
//                       unsigned regind) ;
// void compile_fin_prgr() ;
// void compile_fin_quest(unsigned arite) ;
//
//-------------------------------------------------------------------------

 unsigned arite,art[l_max],i,ils,itabl,itfc,j,nbp,position,sl;
 ttablisteS tablisteS[arite_max];
 tlisteS listeS[l_max];

 ils=0;
 lire();
 strcpy(tabfonct[itf].nom,ch);
 if (c == 'p')
   label[ilbl].type='t';
 label[ilbl].indice=itf;
 label[ilbl].etiqt=P;
 ilbl++;
 itf++;
 lire();
 if (ch[0] == '(')
 { while (ch[0] != ')')
   { position=lettre;
     lire();
     if ((64<ch[0])&&(ch[0]<91))
     { listeS[ils].type=0;
           listeS[ils].pos=mettre_var(ils+1);
           listeS[ils].reg=ils+1;
           ils++;
           lire();
           if ((ch[0] != ')')&&(ch[0] != ','))
             sortie_erreur(1);

     }
     else
     { listeS[ils].pos=position;
           listeS[ils].reg=ils+1;
           ils++;
           lire();
```

```
          if ((ch[0] == ')')||(ch[0] == ','))
            listeS[ils-1].type=2;
          else
          { listeS[ils-1].type=1;
            if ((ch[0] != '('))
              break;
            else
            { nbp=1;
              do
              { lire();
                switch(ch[0])
                { case '(' : nbp++;break;
                      case ')' : nbp--;break;
                }
              }
              while (nbp != 0);
              lire();
              if ((ch[0] != ')')&&(ch[0] != ','))
                sortie_erreur(1);
            }
          }
      }
    }
  }
  if (c == 'p')
    compile_debut_prgr(ils);
  index=index_max;
  if ((ch[0]!= '.')&&(ch[0]!=',')&&(ch[0]!=':'))
    sl=lettre;
  else
  {
   sl = lettre-1 ;
  }
  i=0;
  itabl=0;
  while (i != ils)
  { if (listeS[i].type == 0)
    { switch(c)
      { case 'p' :
              { compile_var_prgr(tabvar[listeS[i].pos].nom,
&tabvar[listeS[i].pos].reg,i+1);
              break;
              }
            case 'q' :
              { compile_var_quest(tabvar[listeS[i].pos].nom,
&tabvar[listeS[i].pos].reg,i+1);
              break;
              }
      }
    }
    else
    { itfc=itf;
      arite=lire_sous_terme(listeS[i].pos,tablisteS[itabl].l);
      art[itabl]=arite;
      switch(c)
      { case 'p' :
            { compile_programme(itfc,tablisteS[itabl].l,arite,listeS[i].reg);
              break;
            }
```

```
            case 'q' :
            { j=0;
              while (j != arite)
              { if ((tablisteS[itabl].l[j].type == 1)||
(tablisteS[itabl].l[j].type == 2))
                  Sa(tablisteS[itabl].l[j].pos,tablisteS[itabl].l[j].reg,c);
                j++;
              }
              compile_question(itfc,tablisteS[itabl].l,arite,listeS[i].reg);
              break;
            }
     }
     itabl++;
   }
   i++;
  }
  if (c == 'p')
  {  i=0;
   while (i != itabl)
   { j=0;
     while(j != art[i])
     { if ((tablisteS[i].l[j].type == 1)||(tablisteS[i].l[j].type == 2))
            Sa(tablisteS[i].l[j].pos,tablisteS[i].l[j].reg,c);
          j++;
     }
     i++;
   }
  }
  switch(c)
  { case 'p' :
    { compile_fin_prgr();
      break;
    }
    case 'q' :
    { compile_fin_quest(ils);
      break;
    }
  }

  lettre=sl;
  lire() ;
  return(ils);
}
```

**unsigned lire_sous_terme(unsigned pos,tlisteS listeS[])**

```
{
//-------------------------------------------------------------------------
//
// Cette fonction stocke dans listeS l'ensembles des parametres qui peuvent
// caracteriser les arguments et leur sous termes a savoir : le nom, le
// type, la position de debut de la sous chaine representant le sous terme
// dans le tableau s et le registre qui lui est alloue.
//
// ParamŠtres :
//
// pos : repere dans le tableau s marquant le debut du terme ... compiler.
// listeS : tableau contenant le type de l'argument, le registre qu'il lui
//              est alloue et l'indice du debut de l'argument dans s.
// lire_sous_terme : arite du sous_terme.
```

```
//
// Uses :
//
// void lire() ;
// unsigned appartient_tabvar(tlisteS listeS[],unsigned ils) ;
// void sortie_erreur(int statut) ;
//
//---------------------------------------------------------------------------

 unsigned ils,nbp,position;

 ils=0;
 lettre=pos;
 lire();
 strcpy(tabfonct[itf].nom,ch);
 itf++;
 lire();
 switch(ch[0])
 { case '(' :
  { while (ch[0] != ')')
    { position=lettre;
          lire();
          if ((64<ch[0])&&(ch[0]<91))
          { listeS[ils].type=0;
           listeS[ils].reg=appartient_tabvar(listeS,ils);
           ils++;
           lire();
           if ((ch[0] != ')')&&(ch[0] != ','))
             sortie_erreur(1);
          }
          else
          { listeS[ils].pos=position;
           listeS[ils].reg=index;
           index++;
           ils++;
           lire();
           if ((ch[0] == ')')||(ch[0] == ','))
             listeS[ils-1].type=2;
           else
           { listeS[ils-1].type=1;
            if ((ch[0] != '('))
              break;
            else
            { nbp=1;
              do
              { lire();
                    switch(ch[0])
                    { case '(' : nbp++;break;
                      case ')' : nbp--;break;
                    }
              }
              while (nbp != 0);
              lire();
              if ((ch[0] != ')')&&(ch[0] != ','))
                    sortie_erreur(1);
            }
           }
          }
    }
```

```
    break ;
  //case ')' : lire() ;
   }
 }
 return(ils);
}

unsigned appartient_tabvar(tlisteS listeS[],unsigned ils)
{
//---------------------------------------------------------------------
//
// Cette fonction alloue aux variables des sous-termes des arguments un
// "bon" registre et met a jour l'ensemble des parametres permettant cette
// action.
//
// Parametres :
//
// listeS : tableau contenant le type de l'argument, le registre qui lui
//              est alloue et l'indice du debut de l'argument dans s.
// ils : indice du tableau listeS qui represente le terme en court de trai-
//      tement.
// appartient_tabvar : contient le numero du registre.
//
// Uses :
//
// unsigned permanent(char nom[lg_var_max]) ;
//
//---------------------------------------------------------------------

 int app,i,trouve;

 i=0;
 trouve=0;
 while ((tabvar[i].nom[0] != '$')&&(trouve == 0))
 { if (strcmp(tabvar[i].nom,ch) == 0)
    trouve=1;
   else
    i++;
 }
 if (trouve == 1)
 { listeS[ils].pos=i;
  if ((tabvar[i].A != 0)&&(tabvar[i].reg == 0))
  { if (permanent(tabvar[i].nom) == 0)
    { tabvar[i].reg=index;
          index++;
          return(index-1);
    }
    else
    { tabvar[i].reg=indexy;
          indexy++;
          return(indexy-1);
    }
  }
  else
    return(tabvar[i].reg);
 }
 else
 { if (permanent(tabvar[i].nom) == 0)
  { strcpy(tabvar[i].nom,ch);
```

```
    tabvar[i].reg=index;
    listeS[ils].pos=i;
    index++;
    return(index-1);
   }
   else
   { strcpy(tabvar[i].nom,ch);
    tabvar[i].reg=indexy;
    listeS[ils].pos=i;
    indexy++;
    return(indexy-1);
   }
  }
}

unsigned mettre_var(unsigned reg)
{
//---------------------------------------------------------------------------
//
// Cette fonction alloue aux arguments qui sont des variables un registre
// et met a jour l'ensemble des parametres permettant cette action.
//
// Parametre :
//
// reg : numero du registre qui doit etre alloue a l'argument.
// mettre_var : contient le numero du registre.
//
//---------------------------------------------------------------------------

 int app,i,trouve;

 i=0;
 trouve=0;
 while ((tabvar[i].nom[0] != '$')&&(trouve == 0))
 { if (strcmp(tabvar[i].nom,ch) == 0)
    trouve=1;
   else
    i++;
 }
 if (trouve == 1)
 { if (tabvar[i].A == 0)
    tabvar[i].A=reg;
   util[reg]=1;
   return(i);
 }
 else
 { strcpy(tabvar[i].nom,ch);
   tabvar[i].A=reg;
   return(i);
 }
}

void compile_debut_quest()
{
//---------------------------------------------------------------------------
//
// Cette fonction assure la compilation du debut des buts en generant le
// code de l'instruction "Allocate".
//
```

```
//---------------------------------------------------------------------

 cprintf("Allocate %u \r\n",tab_clause[i].N);
 Mem.Code[P].cel.nom=21;
 Mem.Code[P].cel.art=2;
 Mem.Code[P+1].cel.nom=tab_clause[i].N;
 P=P+2;
}

void compile_question(unsigned itfc,tlisteS listeS[],unsigned arite, unsigned
regind)
{
//---------------------------------------------------------------------
//
// Cette fonction assure la compilation des differents arguments ( qui ne
// sont pas des variables) des buts, genere un code interpretable et met a
// jour les structures de donnee utiles a cet effet.
//
// Parametres :
//
// itfc : pointeur vers la table des foncteurs.
// listeS : tableau contenant le type de l'argument, le registre qui lui
//          alloue et l'indice du debut de l'argument dans s.
// arite : arite du foncteur.
// regind : contient le numero du registre de l'argument.
//
// Uses :
//
// unsigned permanent(char nom[lg_var_max]) ;
//
//---------------------------------------------------------------------

 int i;

 cprintf("Put_Structure %s/%d X%d \r\n",tabfonct[itfc].nom,arite,regind);
 Mem.Code[P].cel.nom=3;
 Mem.Code[P].cel.art=3;
 Mem.Code[P+1].cel.nom=itfc;
 Mem.Code[P+1].cel.art=arite;
 Mem.Code[P+2].cel.nom=regind;
 P=P+3;
 i=0;
 while (i != arite)
 { switch(listeS[i].type)
  { case 0 :
    { if ((util[listeS[i].reg] == 0)&&(util[tabvar[listeS[i].pos].A]==0))
          { util[listeS[i].reg]=1;
           if (tabvar[listeS[i].pos].A != 0)
             util[tabvar[listeS[i].pos].A]=1;
           if (permanent(tabvar[listeS[i].pos].nom) == 0)
           {
             cprintf("Set_Variable X%d \r\n",listeS[i].reg);
             Mem.Code[P].cel.nom=13;
             Mem.Code[P].cel.art=2;
             Mem.Code[P+1].cel.nom=listeS[i].reg;
             P=P+2;
           }
           else
           {
```

```
            cprintf("Set_Variable Y%d \r\n",listeS[i].reg);
            Mem.Code[P].cel.nom=14;
            Mem.Code[P].cel.art=2;
            Mem.Code[P+1].cel.nom=listeS[i].reg;
            P=P+2;
           }
          }
          else
          { if (permanent(tabvar[listeS[i].pos].nom) == 0)
           {
            cprintf("Set_Value X%d \r\n",listeS[i].reg);
            Mem.Code[P].cel.nom=15;
            Mem.Code[P].cel.art=2;
            Mem.Code[P+1].cel.nom=listeS[i].reg;
            P=P+2;
           }
           else
           {
            cprintf("Set_Value Y%d \r\n",listeS[i].reg);
            Mem.Code[P].cel.nom=16;
            Mem.Code[P].cel.art=2;
            Mem.Code[P+1].cel.nom=listeS[i].reg;
            P=P+2;
           }
          }
          break;
    }
    default :
    { cprintf("Set_Value X%d \r\n",listeS[i].reg);
          Mem.Code[P].cel.nom=15;
          Mem.Code[P].cel.art=2;
          Mem.Code[P+1].cel.nom=listeS[i].reg;
          P=P+2;
    }
   }
   i++;
  }
}

void compile_var_quest(char nom[lg_var_max],unsigned *regx,unsigned rega)
{
//---------------------------------------------------------------------------
//
// Cette fonction assure la compilation des differents arguments ( qui
// sont des variables ) des buts, genere un code interpretable et met a
// jour les structures de donnee utiles a cet effet.
//
// Parametres :
//
// nom : tableau contenant le nom de la variable.
// *regx : parametre passe par adresse qui contient le registre permanent
//         de la variable.
// rega : contient le registre temporaire de la variable.
//
// Uses :
//
// unsigned permanent(char nom[lg_var_max]) ;
//
//---------------------------------------------------------------------------
```

```
if (util[rega] == 0)
{ if (permanent(nom) == 0)
  { if (*regx == 0)
    { *regx=index;
          index++;
    }
    cprintf("Put_Variable X%d,A%d \r\n",*regx,rega);
    util[rega]=1;
    Mem.Code[P].cel.nom=4;
    Mem.Code[P].cel.art=2;
          /* REM : on ne connait pas encore la valeur de regx */
    Mem.Code[P+1].cel.nom=*regx;
    Mem.Code[P+1].cel.art=rega;
    P=P+2;
  }
  else
  { if (*regx == 0)
    { *regx=indexy;
          indexy++;
    }
    cprintf("Put_Variable Y%d,A%d \r\n",*regx,rega);
    util[rega]=1;
    Mem.Code[P].cel.nom=5;
    Mem.Code[P].cel.art=2;
          /* REM : on ne connait pas encore la valeur de regx */
    Mem.Code[P+1].cel.nom=*regx;
    Mem.Code[P+1].cel.art=rega;
    P=P+2;
  }
}
else
{ if (permanent(nom) == 0)
  { if (*regx == 0)
    { *regx=index;
          index++;
    }
    cprintf("Put_Value X%d,A%d \r\n",*regx,rega);
    Mem.Code[P].cel.nom=6;
    Mem.Code[P].cel.art=2;
    Mem.Code[P+1].cel.nom=*regx;
    Mem.Code[P+1].cel.art=rega;
    P=P+2;
  }
  else
  { if (*regx == 0)
    { *regx=indexy;
          indexy++;
    }
    cprintf("Put_Value Y%d,A%d \r\n",*regx,rega);
    Mem.Code[P].cel.nom=7;
    Mem.Code[P].cel.art=2;
    Mem.Code[P+1].cel.nom=*regx;
    Mem.Code[P+1].cel.art=rega;
    P=P+2;
  }
 }
}
```

```
void compile_fin_quest(unsigned arite)
{
//-------------------------------------------------------------------------
//
// Cette fonction assure la compilation de la fin des buts en generant le
// code de l'instruction "Call".
//
// Parametres :
//
// arite : arite du foncteur.
//
//-------------------------------------------------------------------------

 int j;
 char trouve;

 index_max=index;
 trouve=0;
 j=0;
 while ((j != ilbl-1)&&(trouve==0))
 { if ((strcmp(tabfonct[label[j].indice].nom,
          tabfonct[label[ilbl-1].indice].nom) == 0)&&(arite ==
          tabfonct[label[j].indice].arite)&&(label[j].type == 't'))
  { trouve=1;
   cprintf("Call %s/%d \r\n",tabfonct[label[j].indice].nom,arite);
   Mem.Code[P+1].cel.nom=label[j].etiqt;
  }
  else j++;
 }
 if (trouve == 0)
  {
  cprintf("reference en avant \r\n") ;
  Mem.Code[P+1].cel.nom=0;
  tab_ref_avant[RA].nom=label[ilbl-1].indice ;
  tab_ref_avant[RA].p = P+1 ;
  tab_ref_avant[RA].art = arite ;
  tab_ref_avant[RA].etiq = ilbl-1 ;
  RA++ ;
  }
 Mem.Code[P].cel.nom=2;
 Mem.Code[P].cel.art=2;
 Mem.Code[P+1].cel.art=label[j].indice;
 P=P+2;
}

void compile_programme(unsigned itfc,tlisteS listeS[],unsigned arite,
          unsigned regind)
{
//-------------------------------------------------------------------------
//
// Cette fonction assure la compilation des differents arguments ( qui ne
// sont pas des variables) des tete de clause, genere un code interpretable
// et met a jour les structures de donnee utiles a cet effet.
//
// Parametres :
//
// itfc : pointeur vers la table des foncteurs.
// listeS : tableau contenant le type de l'argument, le registre qui lui
//          alloue et l'indice du debut de l'argument dans s.
```

```
// arite : ariteedu foncteur.
// regind : contient le numero du registre de l'argument.
//
// Uses :
//
// unsigned permanent(char nom[lg_var_max]) ;
//
//-----------------------------------------------------------------------

 int i;

 cprintf("Get_Structure %s/%d X%d \r\n",tabfonct[itfc].nom,arite,regind);
 Mem.Code[P].cel.nom=8;
 Mem.Code[P].cel.art=3;
 Mem.Code[P+1].cel.nom=itfc;
 Mem.Code[P+1].cel.art=arite;
 Mem.Code[P+2].cel.nom=regind;
 P=P+3;
 i=0;
 while (i != arite)
 { switch( listeS[i].type)
  { case 0 :
    { if ((util[listeS[i].reg] == 0)&&(util[tabvar[listeS[i].pos].A]==0))
          { util[listeS[i].reg]=1;
           if (tabvar[listeS[i].pos].A != 0)
            util[tabvar[listeS[i].pos].A]=1;
           if (permanent(tabvar[listeS[i].pos].nom) == 0)
           { cprintf("Unify_Variable X%d \r\n",listeS[i].reg);
            Mem.Code[P].cel.nom=17;
            Mem.Code[P].cel.art=2;
            Mem.Code[P+1].cel.nom=listeS[i].reg;
            P=P+2;
           }
           else
           { cprintf("Unify_Variable Y%d \r\n",listeS[i].reg);
            Mem.Code[P].cel.nom=18;
            Mem.Code[P].cel.art=2;
            Mem.Code[P+1].cel.nom=listeS[i].reg;
            P=P+2;
           }
          }
          else
          { if (permanent(tabvar[listeS[i].pos].nom) == 0)
           { cprintf("Unify_Value X%d \r\n",listeS[i].reg);
            Mem.Code[P].cel.nom=19;
            Mem.Code[P].cel.art=2;
            Mem.Code[P+1].cel.nom=listeS[i].reg;
            P=P+2;
           }
           else
           { cprintf("Unify_Value Y%d \r\n",listeS[i].reg);
            Mem.Code[P].cel.nom=20;
            Mem.Code[P].cel.art=2;
            Mem.Code[P+1].cel.nom=listeS[i].reg;
            P=P+2;
           }
          }
          break;
    }
```

```
    default :
    { cprintf("Unify_variable X%d \r\n",listeS[i].reg);
          Mem.Code[P].cel.nom=17;
          Mem.Code[P].cel.art=2;
          Mem.Code[P+1].cel.nom=listeS[i].reg;
          P=P+2;
    }
   }
   i++;
  }
}

void compile_debut_prgr(unsigned arite)
{
//---------------------------------------------------------------------------
//
// Cette fonction assure la compilation du debut des tetes de clause en
// generant le code des instructions "Trust me", "Retry me", "Retry me     // else" et "Allocate".
//
// Parametres :
//
// arite : arite du foncteur.
//
// Uses :
//
// void Affiche_erreur(char str[80]);
//
//---------------------------------------------------------------------------

 char trouve;
 unsigned j;

// retrouver l'indice du foncteur dans tab_tete

 trouve=0;
 j=0;
 while (( j < TT )&&(trouve == 0))
 {
  if ((strcmp(tabfonct[label[ilbl-1].indice].nom,tab_tete[j].nom) == 0)
  && (arite == tab_tete[j].art))
   trouve = 1;
  else
   j++;
 }
 if (trouve == 0)
  Affiche_Erreur("le foncteur de tete est absent");

 cprintf("%s/%d : \r\n",tabfonct[label[ilbl-1].indice].nom,arite);

 if (tab_tete[j].nbocc == 1)
 {
  if (tab_tete[j].etiq >= 0)
  { cprintf("Trust_me %s/%d,l%u \r\n",tab_tete[j].nom,tab_tete[j].art,i);
    Mem.Code[tab_tete[j].etiq].cel.nom=P;
    Mem.Code[P].cel.nom=25;
    Mem.Code[P].cel.art=1;
    P=P+1;
  }
 }
```

```
// nbocc > 1
 else
 {
  if (tab_tete[j].etiq >= 0)
  { cprintf("Retry_Me_Else %s/%d,l%u \r\n",tab_tete[j].nom,                    tab_tete[j].art,i);
   Mem.Code[tab_tete[j].etiq].cel.nom=P;
   tab_tete[j].nbocc--;
   tab_tete[j].etiq=P+1;
   Mem.Code[P].cel.nom=24;
   Mem.Code[P].cel.art=2;
   Mem.Code[P+1].mot=0;
   P=P+2;
  }
  else
  { cprintf("Try_Me_Else %s/%d,l%u \r\n",tab_tete[j].nom,                    tab_tete[j].art,i);
   tab_tete[j].nbocc--;
   tab_tete[j].etiq=P+1;
   Mem.Code[P].cel.nom=23;
   Mem.Code[P].cel.art=2;
   Mem.Code[P+1].mot=0;
   P=P+2;
  }
 }
 if (tab_clause[i].fait == 0)
 { cprintf("Allocate %u \r\n",tab_clause[i].N);
  Mem.Code[P].cel.nom=21;
  Mem.Code[P].cel.art=2;
  Mem.Code[P+1].cel.nom=tab_clause[i].N;
  P=P+2;
 }
}
```

**void compile_var_prgr(char nom[lg_var_max],unsigned *regx,unsigned rega)**

```
{
//---------------------------------------------------------------------------
//
// Cette fonction assure la compilation des differents arguments ( qui
// sont des variables ) des tetes de clause, genere un code interpretable
// et met a jour les structures de donnee utiles a cet effet.
//
// Parametres :
//
// nom : tableau contenant le nom de la variable.
// *regx : parametre passe par adresse qui contient le registre permanent
//         de la variable.
// rega : contient le registre temporaire de la variable.
//
// Uses :
//
// unsigned permanent(char nom[lg_var_max]) ;
//
//---------------------------------------------------------------------------

 if (util[rega] == 0)
 { if (permanent(nom) == 0)
   {if (*regx == 0)
    { *regx=index;
          index++;
    }
```

```
    cprintf("Get_Variable X%d,A%d \r\n",*regx,rega);
    util[rega]=1;
    Mem.Code[P].cel.nom=9;
    Mem.Code[P].cel.art=2;
    Mem.Code[P+1].cel.nom=*regx;
    Mem.Code[P+1].cel.art=rega;
    P=P+2;
   }
   else
   {if (*regx == 0)
    { *regx=indexy;
         indexy++;
    }
    cprintf("Get_Variable Y%d,A%d \r\n",*regx,rega);
    util[rega]=1;
    Mem.Code[P].cel.nom=10;
    Mem.Code[P].cel.art=2;
    Mem.Code[P+1].cel.nom=*regx;
    Mem.Code[P+1].cel.art=rega;
    P=P+2;
   }
  }
  else
  { if (permanent(nom) == 0)
   { if (*regx == 0)
    { *regx=index;
         index++;
    }
    cprintf("Get_Value X%d,A%d \r\n",*regx,rega);
    Mem.Code[P].cel.nom=11;
    Mem.Code[P].cel.art=2;
    Mem.Code[P+1].cel.nom=*regx;
    Mem.Code[P+1].cel.art=rega;
    P=P+2;
   }
   else
   { if (*regx == 0)
    { *regx=indexy;
         indexy++;
    }
    cprintf("Get_Value Y%d,A%d \r\n",*regx,rega);
    Mem.Code[P].cel.nom=12;
    Mem.Code[P].cel.art=2;
    Mem.Code[P+1].cel.nom=*regx;
    Mem.Code[P+1].cel.art=rega;
    P=P+2;
   }
  }
}

void compile_fin_prgr()
{
//---------------------------------------------------------------------------
// Cette fonction met a jour le nombre de registres qu'il faudra utiliser
// dans la compilation des buts.
//---------------------------------------------------------------------------

  index_max = index;
}
```

```
char execution_occur()
{
//-------------------------------------------------------------------------
// Cette fonction execute les instructions WAM situees dans le CODE. Elle
// realise l'unification de la question avec l'ensemble des clauses du     // prog.
//
// Parametres :
//
// execution : flag a 1 si une unification a echoue ou si une reference
//             n'existe pas.
//
// Uses :
//
// void W3_put_structure(unsigned nom,unsigned arite,unsigned reg);
// void W3_set_variable_permanent(unsigned regy);
// void W3_set_variable_temporary(unsigned regx);
// void W3_set_value_permanent(unsigned regy);
// void W3_set_value_temporary(unsigned regx);
// char W3_get_structure_occur(unsigned nom,unsigned arite,unsigned reg);
// void W3_unify_variable_permanent(unsigned regy);
// void W3_unify_variable_temporary(unsigned regx);
// char W3_unify_value_permanent_occur(unsigned regy);
// char W3_unify_value_temporary_occur(unsigned regx);
// void W3_put_variable_permanent(unsigned regy,unsigned rega);
// void W3_put_variable_temporary(unsigned regx,unsigned rega);
// void W3_put_value_permanent(unsigned regy,unsigned rega);
// void W3_put_value_temporary(unsigned regx,unsigned rega);
// void W3_get_variable_permanent(unsigned regy,unsigned rega);
// void W3_get_variable_temporary(unsigned regx,unsigned rega);
// char W3_get_value_permanent_occur(unsigned regy,unsigned rega);
// char W3_get_value_temporary_occur(unsigned regx,unsigned rega);
// char W3_call(unsigned nom);
// void W3_proceed();
// void W3_allocate(unsigned n) ;
// void W3_deallocate() ;
// void W3_try_me_else(unsigned L) ;
// void W3_retry_me_else(unsigned L) ;
// void W3_trust_me() ;
// void W3_unwind_trail( long a1, long a2 ) ;
// void W3_trail(unsigned long a) ;
// char backtrack() ;
// void sortie_resultats() ;
// void sauve_contenu_arg() ;
// void maj_ind_arg() ;
//
//-------------------------------------------------------------------------

unsigned  adrfc,arite,cop,fin,reg,P_max;
char echec ;

fin = 0 ;
echec = 0 ;
P_max=P_debut;

do
{
 cop = Mem.Code[P].cel.nom ;
 switch( cop )
```

```
{
 case 0 : fin = 1 ;
          break ;
 case 1 : W3_proceed() ;
          break ;
 case 2 : if (P >= P_debut)
          { if (P > P_max)
           { P_max=P;
                sauve_contenu_arg();
           }
          }
          echec=W3_call(Mem.Code[P+1].cel.nom) ;
          break ;
 case 3 : W3_put_structure(Mem.Code[P+1].cel.nom,Mem.Code[P+1].cel.art,Mem.Code[P+2].cel.nom);
          P=P+3 ;
          break ;
 case 4 : W3_put_variable_temporary(Mem.Code[P+1].cel.nom,Mem.Code[P+1].cel.art) ;
          P = P+2 ;
          break ;
 case 5 : W3_put_variable_permanent(Mem.Code[P+1].cel.nom,Mem.Code[P+1].cel.art) ;
          P = P+2 ;
          break ;
 case 6 : W3_put_value_temporary(Mem.Code[P+1].cel.nom,Mem.Code[P+1].cel.art) ;
          P = P+2 ;
          break ;
 case 7 : W3_put_value_permanent(Mem.Code[P+1].cel.nom,Mem.Code[P+1].cel.art) ;
          P = P+2 ;
          break ;
 case 8 :
echec=W3_get_structure_occur(Mem.Code[P+1].cel.nom,Mem.Code[P+1].cel.art,Mem.Code[P+2].cel.nom);
          if (echec==1)
           echec = backtrack() ;
          else
           P=P+3 ;
          break ;
 case 9 : W3_get_variable_temporary(Mem.Code[P+1].cel.nom,Mem.Code[P+1].cel.art);
          P=P+2;
          break;
 case 10: W3_get_variable_permanent(Mem.Code[P+1].cel.nom,Mem.Code[P+1].cel.art);
          P=P+2;
          break;
 case 11: echec=W3_get_value_temporary_occur(Mem.Code[P+1].cel.nom,Mem.Code[P+1].cel.art);
          if (echec==1)
           echec = backtrack() ;
          else
           P=P+2;
          break;
 case 12: echec=W3_get_value_permanent_occur(Mem.Code[P+1].cel.nom,Mem.Code[P+1].cel.art);
          if (echec==1)
           echec = backtrack() ;
          else
           P=P+2;
          break;
 case 13: W3_set_variable_temporary(Mem.Code[P+1].cel.nom);
          P=P+2;
          break;
 case 14: W3_set_variable_permanent(Mem.Code[P+1].cel.nom);
          P=P+2;
          break;
```

```
 case 15: W3_set_value_temporary(Mem.Code[P+1].cel.nom) ;
          P=P+2 ;
          break ;
 case 16: W3_set_value_permanent(Mem.Code[P+1].cel.nom) ;
          P=P+2 ;
          break ;
 case 17: W3_unify_variable_temporary(Mem.Code[P+1].cel.nom) ;
          P=P+2 ;
          break ;
 case 18: W3_unify_variable_permanent(Mem.Code[P+1].cel.nom);
          P=P+2;
          break;
 case 19: echec=W3_unify_value_temporary_occur(Mem.Code[P+1].cel.nom);
          if (echec==1)
            echec = backtrack() ;
          else
            P=P+2 ;
          break ;
 case 20: echec=W3_unify_value_permanent_occur(Mem.Code[P+1].cel.nom);
          if (echec==1)
            echec = backtrack() ;
          else
            P=P+2 ;
          break ;
 case 21: W3_allocate(Mem.Code[P+1].cel.nom) ;
          P=P+2;
          break;
 case 22: W3_deallocate() ;
          if ((P > P_debut)&&(B > heap_max))
          { sortie_resultats();
            maj_ind_arg();
            P_max=P_debut;
            echec=backtrack();
          }
          else
            P=P+1 ;
          break;
 case 23: W3_try_me_else(Mem.Code[P+1].cel.nom);
          P=P+2;
          break;
 case 24: W3_retry_me_else(Mem.Code[P+1].cel.nom);
          P=P+2;
          break;
 case 25: W3_trust_me() ;
          P=P+1;
          break;
 default : Affiche_Erreur("Code operatoire incorrect") ;
           echec = 1 ;
 }
}
while ((fin == 0)&&(echec==0)) ;

return(echec);
}

char execution()
{
//-------------------------------------------------------------------------
// Cette fonction execute les instructions WAM situees dans le CODE. Elle
```

```
// realise l'unification de la question avec l'ensemble des clauses du    // prog.
//
// Parametres :
//
// execution : flag a 1 si une unification a echoue ou si une reference
//             n'existe pas.
//
// Uses :
//
// void W3_put_structure(unsigned nom,unsigned arite,unsigned reg);
// void W3_set_variable_permanent(unsigned regy);
// void W3_set_variable_temporary(unsigned regx);
// void W3_set_value_permanent(unsigned regy);
// void W3_set_value_temporary(unsigned regx);
// char W3_get_structure(unsigned nom,unsigned arite,unsigned reg);
// void W3_unify_variable_permanent(unsigned regy);
// void W3_unify_variable_temporary(unsigned regx);
// char W3_unify_value_permanent(unsigned regy);
// char W3_unify_value_temporary(unsigned regx);
// void W3_put_variable_permanent(unsigned regy,unsigned rega);
// void W3_put_variable_temporary(unsigned regx,unsigned rega);
// void W3_put_value_permanent(unsigned regy,unsigned rega);
// void W3_put_value_temporary(unsigned regx,unsigned rega);
// void W3_get_variable_permanent(unsigned regy,unsigned rega);
// void W3_get_variable_temporary(unsigned regx,unsigned rega);
// char W3_get_value_permanent(unsigned regy,unsigned rega);
// char W3_get_value_temporary(unsigned regx,unsigned rega);
// char W3_call(unsigned nom);
// void W3_proceed();
// void W3_allocate(unsigned n) ;
// void W3_deallocate() ;
// void W3_try_me_else(unsigned L) ;
// void W3_retry_me_else(unsigned L) ;
// void W3_trust_me() ;
// void W3_unwind_trail( long a1, long a2 ) ;
// void W3_trail(unsigned long a) ;
// char backtrack() ;
// void sortie_resultats() ;
// void sauve_contenu_arg() ;
// void maj_ind_arg() ;
//
//-------------------------------------------------------------------------

unsigned  adrfc,arite,cop,fin,reg,P_max;
char echec ;

fin = 0 ;
echec = 0 ;
P_max=P_debut;

do
{
 cop = Mem.Code[P].cel.nom ;
 switch( cop )
 {
  case 0 : fin = 1 ;
           break ;
  case 1 : W3_proceed() ;
           break ;
```

```
case 2 : if (P >= P_debut)
          { if (P > P_max)
            { P_max=P;
                sauve_contenu_arg();
            }
          }
          echec=W3_call(Mem.Code[P+1].cel.nom) ;
          break ;
case 3 : W3_put_structure(Mem.Code[P+1].cel.nom,Mem.Code[P+1].cel.art,Mem.Code[P+2].cel.nom);
          P=P+3 ;
          break ;
case 4 : W3_put_variable_temporary(Mem.Code[P+1].cel.nom,Mem.Code[P+1].cel.art) ;
          P = P+2 ;
          break ;
case 5 : W3_put_variable_permanent(Mem.Code[P+1].cel.nom,Mem.Code[P+1].cel.art) ;
          P = P+2 ;
          break ;
case 6 : W3_put_value_temporary(Mem.Code[P+1].cel.nom,Mem.Code[P+1].cel.art) ;
          P = P+2 ;
          break ;
case 7 : W3_put_value_permanent(Mem.Code[P+1].cel.nom,Mem.Code[P+1].cel.art) ;
          P = P+2 ;
          break ;
case 8 : echec=W3_get_structure(Mem.Code[P+1].cel.nom,Mem.Code[P+1].cel.art,Mem.Code[P+2].cel.nom);
          if (echec==1)
            echec = backtrack() ;
          else
            P=P+3 ;
          break ;
case 9 : W3_get_variable_temporary(Mem.Code[P+1].cel.nom,Mem.Code[P+1].cel.art);
          P=P+2;
          break;
case 10: W3_get_variable_permanent(Mem.Code[P+1].cel.nom,Mem.Code[P+1].cel.art);
          P=P+2;
          break;
case 11: echec=W3_get_value_temporary(Mem.Code[P+1].cel.nom,Mem.Code[P+1].cel.art);
          if (echec==1)
            echec = backtrack() ;
          else
            P=P+2;
          break;
case 12: echec=W3_get_value_permanent(Mem.Code[P+1].cel.nom,Mem.Code[P+1].cel.art);
          if (echec==1)
            echec = backtrack() ;
          else
            P=P+2;
          break;
case 13: W3_set_variable_temporary(Mem.Code[P+1].cel.nom);
          P=P+2;
          break;
case 14: W3_set_variable_permanent(Mem.Code[P+1].cel.nom);
          P=P+2;
          break;
case 15: W3_set_value_temporary(Mem.Code[P+1].cel.nom) ;
          P=P+2 ;
          break ;
case 16: W3_set_value_permanent(Mem.Code[P+1].cel.nom) ;
          P=P+2 ;
          break ;
```

```
  case 17: W3_unify_variable_temporary(Mem.Code[P+1].cel.nom) ;
           P=P+2 ;
           break ;
  case 18: W3_unify_variable_permanent(Mem.Code[P+1].cel.nom);
           P=P+2;
           break;
  case 19: echec=W3_unify_value_temporary(Mem.Code[P+1].cel.nom);
           if (echec==1)
            echec = backtrack() ;
           else
            P=P+2 ;
           break ;
  case 20: echec=W3_unify_value_permanent(Mem.Code[P+1].cel.nom);
           if (echec==1)
            echec = backtrack() ;
           else
            P=P+2 ;
           break ;
  case 21: W3_allocate(Mem.Code[P+1].cel.nom) ;
           P=P+2;
           break;
  case 22: W3_deallocate() ;
           if ((P > P_debut)&&(B > heap_max))
           { sortie_resultats();
            maj_ind_arg();
            P_max=P_debut;
            echec=backtrack();
           }
           else
            P=P+1 ;
           break;
  case 23: W3_try_me_else(Mem.Code[P+1].cel.nom);
           P=P+2;
           break;
  case 24: W3_retry_me_else(Mem.Code[P+1].cel.nom);
           P=P+2;
           break;
  case 25: W3_trust_me() ;
           P=P+1;
           break;
  default : Affiche_Erreur("Code operatoire incorrect") ;
            echec = 1 ;
 }
 }
 while ((fin == 0)&&(echec==0)) ;

 return(echec);
}

void init_tabvar()
{
//-------------------------------------------------------------------------
//
// Cette fonction initialise le tableau tabvar.
//
//-------------------------------------------------------------------------

 int i;
```

```
 i=0;
 while (i != nb_var)
 { tabvar[i].nom[0]='$';
  tabvar[i].reg=0;
  tabvar[i].A=0;
  i++;
 }
}

void init_tab_ref_avant()
{
//---------------------------------------------------------------------
//
// Cette fonction initialise le tableau tab_ref_avant.
//
//---------------------------------------------------------------------

 int i;

 i=0;
 while (i < ref_max)
 { tab_ref_avant[i].nom=0;
  tab_ref_avant[i].art=0;
  tab_ref_avant[i].p=0;
  tab_ref_avant[i].etiq=0;
  i++;
 }
}

void init_util()
{
//---------------------------------------------------------------------
//
// Cette fonction initialise le tableau util.
//
//---------------------------------------------------------------------

 int i;

 i=0;
 while (i != arite_max)
 { util[i]=0;
  i++;
 }
}

void init_argument()
{
//---------------------------------------------------------------------
//
// Cette fonction initialise une partie du tableau tabvar, a savoir les
// champs "nom" et "A".
//
//---------------------------------------------------------------------

 unsigned i;

 i=0;
 while(tabvar[i].nom[0] != '$')
```

```
 { tabvar[i].A=0;
  i++;
 }
}

void sortie_erreur(int statut)
{
//---------------------------------------------------------------------
//
// Cette fonction permet l'impression de divers messages d'erreur.
//
// Parametre
//
// statut : code erreur.
//
// Uses :
//
// void Affiche_erreur(char str[80]) ;
//
//---------------------------------------------------------------------

 switch(statut)
 { case 1 : Affiche_Erreur("mauvaise syntaxe sur ')' ou ',' \n") ;
            break;
  case 2 : Affiche_Erreur("mauvaise syntaxe sur ':' ou '-' \n") ;
            break;
  case 3 : Affiche_Erreur("mauvaise syntaxe sur ',' ou '.' \n") ;
            break;
 }
 exit(1);
}

void decode_terme (tcellule cel)
{
//---------------------------------------------------------------------
// Cette fonction decode le terme represente par la cellule cel. Cette
// cellule possede deux champs : son adresse et son type. Suivant le type
// du terme, on reconstituera la variable ou la structure comme il se doit.
//
// Parametre :
//
// cel : cellule representant le terme a reconstituer.
//
// Uses :
//
// unsigned long deref (unsigned long adr) ;
// void decode_foncteur (tcellule cel) ;
// void decode_terme (tcellule cel) ;
// void adresse (unsigned long mot) ;
// void type (unsigned long mot) ;
//
//---------------------------------------------------------------------

 tcellule cel1,cel2 ;
 unsigned arite,i ;

 if ((type(cel.mot)==0)&&(type(Mem.Heap[Deref(adresse(cel.mot))].mot)==0))
  {
   cel1 = Mem.Heap[Deref(adresse(cel.mot))] ;
```

```
   cprintf("Var_%u",adresse(cel1.mot));
  }
 else
  {
  if (type(cel.mot) ==1)
    cel1 = Mem.Heap[adresse(cel.mot)] ;
  else
   {
    cel = Mem.Heap[Deref(adresse(cel.mot))] ;
    cel1 = Mem.Heap[adresse(cel.mot)] ;
   }
  arite = decode_foncteur (cel1) ;
  if ( arite > 0 )
    {
     cprintf("(") ;
     for ( i = adresse(cel.mot)+1 ; i <= adresse(cel.mot)+arite ; i++ )
           {
           cel2 = Mem.Heap[i] ;
           decode_terme ( cel2 ) ;
           if (i<adresse(cel.mot)+arite) printf(",") ;
           }
     cprintf(")") ;
    }
  }
}

unsigned decode_foncteur (tcellule cel)
{
//---------------------------------------------------------------------
// Cette fonction renvoit l'arite du foncteur et imprime le foncteur.
//
// Parametres :
//
// cel : cellule dont on cherche le nom du foncteur et son arite.
// decode_foncteur : arite du terme.
//
//---------------------------------------------------------------------

 cprintf(tabfonct[cel.cel.nom].nom) ;
 return(cel.cel.art) ;
}

unsigned permanent(char nom[lg_var_max])
{
//---------------------------------------------------------------------
//
// Cette fonction retourne la valeur 1 si la variable de nom "nom" est une
// variable permanente.
//
// Parametre :
//
// nom : contient le nom de la variable.
// permanent : flag a 1 si la variable 'nom' est permanente.
//
//---------------------------------------------------------------------

 unsigned j;

 j=0;
```

```
 while(strcmp(nom,tab_clause[i].variables[j].nom) != 0)
  j++;
 return(tab_clause[i].variables[j].vu);
}



unsigned compile_clause(char c)
{
//---------------------------------------------------------------------------
//
// Cette fonction permet la compilation des clauses d'un programme.
//
// parametre :
//
// c : flag permettant de savoir si la clause a compiler est une question
//     ou une clause du programme.
// compile_clause : pointeur vers la table des foncteurs.
//
// Uses :
//
// void init_tabvar() ;
// void init_util() ;
// unsigned lire_terme(char c) ;
// void touche() ;
// void lire() ;
// void sortie_erreur(int statut) ;
// void compile_debut_quest() ;
// void init_argument() ;
//
//---------------------------------------------------------------------------

 unsigned itfc;

 index=1;
 indexy=1;
 index_max=(tab_clause[i].art_max) + 1;
 lettre=0;
 init_tabvar();

 if (c == 'p')
 { init_util();
  itfc=itf;
  tabfonct[itfc].arite=lire_terme('p');
  touche() ;
  switch(ch[0])
  { case ':' : lire();
                    if (ch[0] != '-')
                      sortie_erreur(2) ;
                    break ;
    case '.' : break ;
    default : sortie_erreur(3) ;
                   break ;
  }
 }
 else
 { ch[0]='n';
  lettre=2;
```

```
   compile_debut_quest();
  }

  while (ch[0] != '.')
  { init_util();
   init_argument();
   itfc=itf;
   tabfonct[itfc].arite=lire_terme('q');
   if ((ch[0] != ',')&&(ch[0] != '.'))
    sortie_erreur(3);
  }
   if(tab_clause[i].fait == 0)
   {
    cprintf("Deallocate \r\n");
    Mem.Code[P].cel.nom=22;
    Mem.Code[P].cel.art=1;
    P=P+1;
   }
 if (c == 'p')
 {
  cprintf("proceed \r\n");
  Mem.Code[P].cel.nom=1;
  Mem.Code[P].cel.art=1;
  P++;
 }
 touche() ; ;
 return(itfc);
}

void sauve_contenu_arg()
{
//---------------------------------------------------------------------------
// Cette fonction sauve les arguments lors de leur initialisation afin de
// les retrouver plus tard pour la reconstitution de la ou les solution(s).
//
//---------------------------------------------------------------------------

 unsigned i;

 // sauve le p_counter
 Argument[ind_arg].mot=P;
 ind_arg++;

 i=1;
 // foncteur de tete et arite sauve
 Argument[ind_arg].cel.nom=Mem.Code[P+1].cel.art;
 Argument[ind_arg].cel.art=tabfonct[Mem.Code[P+1].cel.art].arite;
 ind_arg++;

 while (i <= tabfonct[Mem.Code[P+1].cel.art].arite)
 { Argument[ind_arg].mot=Mem.X[i].mot;
  i++;
  ind_arg++;
 }
}

void maj_references_avant()
{
//---------------------------------------------------------------------------
```

```
// Cette fonction met a jour les references en avant qui se sont presentees
// lors de la premiere passe du programme. Elle signale les references    // mises
// a jour et celles qui sont restees incompletes.
//
// Uses :
//
// void Affiche_erreur(char str[80]) ;
//
//-------------------------------------------------------------------------

  unsigned j,k;
  char trouve;

  j=0 ;
  while (j<RA)
  {
   trouve = 0 ;
   k = tab_ref_avant[j].etiq+1 ;
   while ((k<=(ilbl-1))&&(trouve==0))
   {
    if ((strcmp(tabfonct[label[k].indice].nom,
                        tabfonct[tab_ref_avant[j].nom].nom)==0)
          &&(tab_ref_avant[j].art==tabfonct[label[k].indice].arite)
          &&(label[k].type=='t'))
     {
      trouve=1 ;
      Mem.Code[tab_ref_avant[j].p].cel.nom=label[k].etiqt;
      cprintf(" MAJ : call %s/%u \r\n",tabfonct[tab_ref_avant[j].nom].nom,tab_ref_avant[j].art) ;
     }
    else
      k++ ;
   }
   if (trouve==0)
   {
    Affiche_Erreur("ref. en avant incomplete");//,tabfonct[tab_ref_avant[j].nom].nom) ;
   }
   j++ ;
  }
}


void init_label()
{
//-------------------------------------------------------------------------
//
// Cette fonction initialise le tableau label.
//
//-------------------------------------------------------------------------
  unsigned j;

  for(j=0;j<nb_tete;j++)
  { label[j].indice=0;
    label[j].etiqt=0;
    label[j].type=0;
  }
}

void sortie_resultats()
{
```

```
//---------------------------------------------------------------------------
//
// Cette fonction imprime les termes instancies de la question.
//
// Uses :
//
// void decode_terme(tcellule cel) ;
// void touche() ;
//
//---------------------------------------------------------------------------

 unsigned j;
 tcellule cel;

 cprintf("oui, ");
 ind_arg=1;
 while ( Argument[ind_arg].cel.nom > 0 )
 {
  j = 1 ;
  cprintf("%s(",tabfonct[Argument[ind_arg].cel.nom].nom);
  i = tabfonct[Argument[ind_arg].cel.nom].arite;
  ind_arg++ ;
  while (j <= i)
  { cel.mot = Argument[ind_arg].mot ;
    decode_terme ( cel ) ;
    if ( j < i )
          cprintf(",");
    j++ ;
    ind_arg++ ;
  }
  cprintf(") ") ;

  ind_arg++;
 }
 cprintf(".\r\n") ;
 touche();
}

void maj_arguments()
{
//---------------------------------------------------------------------------
//
// Cette fonction initialise le tableau Argument.
//
//---------------------------------------------------------------------------

 unsigned j;

 for (j=0; j<arite_max ; j++)
 {
  Argument[j].mot = 0 ;
 }
}

void maj_ind_arg()
{
//---------------------------------------------------------------------------
//
// Cette fonction met a jour ind_arg, qui est assimile a un pointeur vers
```

```
// une pile. Cette pile est le tableau Argument.
//
//-------------------------------------------------------------------------

 int j,k;

 //rechercher l'indice du p_counter
 j=0;
 k=-1;
 while ((j < arite_max)&&(k == -1))
// le '+2' dans le test vu que l'on sauve lors d'un Call et que l'on revient
// a l'instruction qui suit le call lors du backtracking.
 { if (Argument[j].mot+2 == P)
     k=j;
   else
     j++;
 }
 if (k == -1)
   k=0;

 // mettre a jour le sommet de la pile.
 ind_arg=k;
}
```

# IOWAM.CPP

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <conio.h>

// fonctions internes
void cadre(int x1,int y1,int x2,int y2) ;
    // création d'un cadre défini par les coordonnées (x1,y1) et (x2,y2)
void fenetre(int x1,int y1,int x2,int y2) ;
    // définition d'une fenêtre de coordonnées (x1,y1) et (x2,y2)
void touche() ;
    // attente de la frappe de la touche <return>
void centrer(int x1, int x2, int y, char nom[80]) ;
    // affichage d'une ligne centrée sur l'ordonnée y

// services offerts
void Affiche_Intro() ;
    // affichage de l'introduction
void Affiche_Wdw_Prog() ;
    // activation de la fenêtre d'affichage du programme Prolog
void Affiche_Wdw_Code() ;
    // activation de la fenêtre d'affichage du Code WAM
void Affiche_Wdw_Sol() ;
    // activation de la fenêtre d'affichage des solutions
char* Affiche_Fichier() ;
    // introduction du nom du fichier contenant le programme Prolog
char* Affiche_Question() ;
    // introduction de la question Prolog
char* Affiche_Boucle_Question() ;
    // choix d'une nouvelle question
char* Affiche_Boucle_Prog() ;
    // choix d'une réexécution du compilateur
void Affiche_Erreur(char str[80]) ;
    // affichage d'une erreur syntaxique ou autre
void Affiche_Bye() ;
    // affichage de fin
```